

AD-A189 647

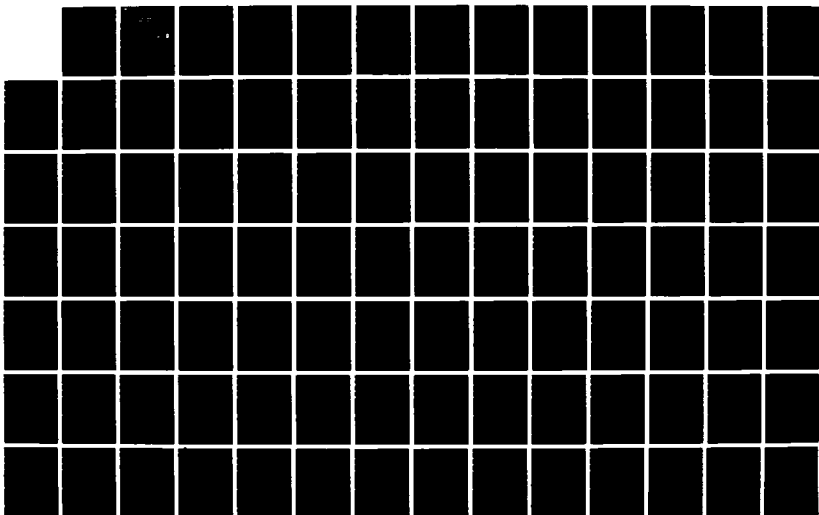
THE ADA (TRADE NAME) COMPILER VALIDATION CAPABILITY
IMPLEMENTERS' GUIDE VERSION 1(U) SOFTECH INC WALTHAM MA
J B GOODENOUGH DEC 86

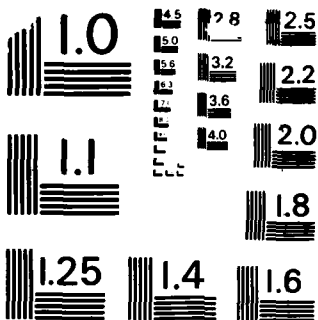
1/9

UNCLASSIFIED

F/G 12/3

NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

DTIC FILE COPY

2

AD-A189 647

The Ada[®] Compiler Validation Capability Implementers' Guide

Version 1

December 1986

John B. Goodenough

SofTech, Inc.

Waltham, MA. 02254-9197

**DTIC
ELECTE
DEC 28 1987**
S D
C&D

DISTRIBUTION STATEMENT A

**Approved for public release;
Distribution Unlimited**

Prepared for the

**ACVC Maintenance Organization
ASD/SIOL
Wright-Patterson AFB OH. 45433-6503**

© Copyright SofTech, Inc., 1986

**® Ada is a registered trademark of the United States Government
(Ada Joint Program Office)**

87 12 14 107

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	12. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) The Ada Compiler Validation Capability Implementers' Guide. Version 1		5. TYPE OF REPORT & PERIOD COVERED December, 1986
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) John B. Goodenough, SofTech, Inc.		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION AND ADDRESS SofTech, Inc. Waltham, MA 02254-9197		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS ACVA Maintenance Organization ASD/SIOL Wright-Patterson AFB OH. 45433-6503		12. REPORT DATE December, 1986
		13. NUMBER OF PAGES 787
14. MONITORING AGENCY NAME & ADDRESS (If different from Controlling Office) Wright-Patterson		15. SECURITY CLASS (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20. If different from Report) UNCLASSIFIED		
18. SUPPLEMENTARY NOTES		
19. KEYWORDS (Continue on reverse side if necessary and identify by block number) Ada Programming language, Ada Compiler Validation, Ada Compiler Validation Capability, ACVC, Validation Testing, Ada Validation Office, AVO, Ada Validation Facility, AVF, ANSI/MIL-STD-1815A, Ada Joint Program Office, AJPO		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) See Attached.		

DD FORM 1473
1 JAN 73

EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Chapter 1

Introduction

➤ The primary purpose of the Ada Compiler Validation Capability (ACVC) is to help decide whether Ada translators conform to ANSI/MIL-STD-1815A-1983, "The Reference Manual for the Ada Language" (the RM). The ACVC has three main components:¹

- An *Implementers' Guide*, which describes implementation implications of the RM and the conditions to be checked by validation tests,
- *Test programs* to be submitted to a compiler, *and*
- *Validation support tools*, which assist in preparing tests for execution and in analyzing the results of executions.

This document is the Ada Implementers' Guide (AIG). Although it follows the structure and numbering of the RM, additional sections have sometimes been created. These sections are designated with letters — e.g., 8.3.a, 8.3.b, etc. — to distinguish them from sections of the RM.

Each numbered/lettered AIG section contains up to seven unnumbered subsections as follows:

1. *Semantic Ramifications* — This subsection documents semantic implications that might not otherwise be obvious from a reading of the RM. When conclusions stated in these subsections are derived from statements in separate sections of the RM, appropriate references are made so the basis for the interpretation is clear. In case of conflict between the AIG and the RM, the RM takes precedence.
2. *Legality Rules* — This subsection explicitly lists context-sensitive syntactic and semantic legality rules to be checked by an Ada translator prior to beginning execution of an Ada program. In essence, all legality conditions other than those expressed by syntax productions in the RM are listed here, even if this listing duplicates wording in the RM. Subtle ways of violating these rules are discussed in the Semantic Ramifications subsection.
3. *Exception Conditions* — This subsection explicitly lists the conditions under which an implementation is required to raise an exception associated with some predefined language construct.
4. *Test Objectives and Design Guidelines* — This subsection specifies the validation tests to be written, lists the problems to keep in mind while writing test cases under "Implementation Guidelines," and, when necessary, outlines the program structure required to satisfy a test objective. At least one test will be written for each objective except when the objective is duplicated by an objective in another section. In such cases, a reference to the implemented objective is given. Test objectives are numbered in increasing order, but not necessarily consecutively.

¹More information about the ACVC can be found in "The Ada Compiler Validation Capability," COMPUTER 14, 6 (June 1981), 57-64 and in "Ada Compiler Validation: An Example of Software Testing Theory and Practice," Technical Report TR-86-07, Wang Institute of Graduate Studies, Tyngsboro, MA, 01824.

5. *Approved Interpretations* — This subsection summarizes approved interpretations of ANSI/MIL-STD-1815A-1983. These interpretations correct errors, ambiguities, or inconsistencies in the RM. Such interpretations are reflected in the validation tests.

In this version of the AIG, only Chapters 2-4 have been fully updated to reflect approved interpretations.

6. *Changes from July 1982* — This subsection briefly describes changes to the draft ANSI/MIL-STD, dated July 1982. This version was distributed widely as part of the standardization process. Changes to the July 1982 version are documented to ensure they are covered by the tests. The changes described in this and the following subsection comprise all the substantive changes made to MIL-STD-1815.
7. *Changes from July 1980* — This subsection briefly describes substantive changes between MIL-STD-1815, July 1980, and the draft ANSI/MIL-STD dated July 1982. These changes are documented to ensure they are covered by the tests.

When referencing a section or paragraph of the RM, the section number is preceded by RM, e.g., RM 12.3.1. To reference a paragraph within a section, the section number is followed by a slash, e.g., RM 12.3/3 means paragraph 3 of section 12.3. A similar method is used to refer to sections of the AIG. For example, IG 12.3/T3 refers to test objective 3 of AIG section 12.3. IG 12.3/S refers to the Semantic Ramifications subsection of AIG section 12.3.

Interpretations of the RM are recommended by the Ada Language Maintenance Panel of the Ada Board and become official (for ANSI/MIL-STD-1815A) when issued by the Director, AJPO (Ada Joint Program Office). These interpretations are contained in *Ada Commentaries*, which are identified by numbers having the form AI-ddddd, e.g., AI-00037. Copies of these Commentaries can be obtained by contacting the AJPO or the Ada Information Clearinghouse. The Commentaries are also available online in the account ADA-COMMENT at ADA20.ISI.EDU and can be mailed electronically to requesters.

Comments on the AIG should be sent to ACVC at ADA20.ISI.EDU using the same format as for comments on the RM:

!section x.y.z(pp) Commenter's Name YY-MM-DD
!version v
!topic brief description of nature of comment

The paragraph numbers should use the numbering provided by the AIG, e.g., 12.3.1(S2) would mean the second Semantic Ramification paragraph of section 12.3.1. When describing typographical errors, use brackets ([]) to indicate what should be deleted and braces ({}) to indicate what should be inserted. Files of AIG comments will be placed in the ADA-LSN account at ADA20.

Table of Contents

1 Introduction	1-1
2 Lexical Elements	2-1
2.1 Character Set	2-1
2.2 Lexical Elements, Separators, and Delimiters	2-2
2.3 Identifiers	2-4
2.4 Numeric Literals	2-5
2.4.1 Decimal Literals	2-6
2.4.2 Based Literals	2-7
2.5 Character Literals	2-8
2.6 String Literals	2-9
2.7 Comments	2-12
2.8 Pragmas	2-13
2.9 Reserved Words	2-17
2.10 Allowable Replacements of Characters	2-18
3 Declarations and Types	3-1
3.1 Declarations	3-1
3.2 Objects and Named Numbers	3-1
3.2.1 Object Declarations	3-3
3.2.2 Number Declarations	3-11
3.3 Types and Subtypes	3-13
3.3.1 Type Declarations	3-16
3.3.2 Subtype Declarations	3-17
3.3.3 Classification of Operations	3-19
3.4 Derived Type Definitions	3-22
3.5 Scalar Types	3-37
3.5.1 Enumeration Types	3-40
3.5.2 Character Types	3-42
3.5.3 Boolean Type	3-43
3.5.4 Integer Types	3-44
3.5.5 Operations of Discrete Types	3-46
3.5.6 Real Types	3-54
3.5.7 Floating Point Types	3-55
3.5.8 Operations of Floating Point Types	3-62
3.5.9 Fixed Point Types	3-63
3.5.10 Operations of Fixed Point Types	3-67
3.6 Array Types	3-69
3.6.1 Index Constraints and Discrete Ranges	3-71
3.6.1.a Discrete Ranges	3-71
3.6.1.b Index Constraints	3-75
3.6.2 Operations of Array Types	3-78
3.6.3 The Type String	3-82



Technically Correct	
DATE	INITIALS
A-1	

Table of Contents

3.7 Record Types	3-84
3.7.1 Discriminants	3-89
3.7.2 Discriminant Constraints	3-93
3.7.3 Variant Parts	3-107
3.7.4 Operations of Record Types	3-110
3.8 Access Types	3-114
3.8.1 Incomplete Type Declarations	3-119
3.8.2 Operations of Access Types	3-125
3.9 Declarative Parts	3-127
 4 Names and Expressions	 4-1
4.1 Names	4-1
4.1.1 Indexed Components	4-3
4.1.2 Slices	4-5
4.1.3 Selected Components	4-8
4.1.4 Attributes	4-18
4.2 Literals	4-22
4.3 Aggregates	4-27
4.3.1 Record Aggregates	4-32
4.3.2 Array Aggregates	4-36
4.4 Expressions	4-53
4.5 Operators and Expression Evaluation	4-55
4.5.1 Logical Operators and Short-Circuit Control Forms	4-57
4.5.1.a Logical Boolean Operators	4-57
4.5.1.b Logical Array Operators	4-58
4.5.1.c Short-circuit Control Forms	4-59
4.5.2 Relational Operators and Membership Tests	4-60
4.5.2.a Relational and Membership Operations (Enumeration)	4-61
4.5.2.b Relational and Membership Operations (Character)	4-62
4.5.2.c Relational and Membership Operations (Boolean)	4-62
4.5.2.d Relational and Membership Operations (Integer)	4-63
4.5.2.e Relational and Membership Operations (Fixed/Float)	4-64
4.5.2.f Relational and Membership Operations (Array)	4-65
4.5.2.g Relational and Membership Operations (Record)	4-67
4.5.2.h Relational and Membership Operations (Access)	4-68
4.5.2.i Relational and Membership Operations (Private/Ltd)	4-70
4.5.3 Binary Adding Operators	4-70
4.5.3.a Integer Adding Operators	4-71
4.5.3.b Floating Point Adding Operators	4-72
4.5.3.c Fixed Point Adding Operators	4-74
4.5.3.d Array Adding Operators (Catenation)	4-74
4.5.4 Unary Adding Operators	4-76
4.5.4.a Integer Unary Adding Operators	4-77
4.5.4.b Real Unary Adding Operators	4-78
4.5.5 Multiplying Operators	4-79
4.5.5.a Integer Multiplying Operators	4-80
4.5.5.b Real Multiplying Operators	4-81

Table of Contents

4.5.6 Highest Precedence Operators	4-84
4.5.6.a Integer Exponentiating Operator	4-85
4.5.6.b Floating Point Exponentiating Operator	4-86
4.5.6.c Integer Absolute Value Operator	4-87
4.5.6.d Real Absolute Value Operator (Fixed/Float)	4-88
4.5.6.e Scalar Negation Operator	4-89
4.5.6.f Array Negation Operator	4-89
4.5.7 Accuracy of Operations with Real Operands	4-89
4.6 Type Conversions	4-90
4.7 Qualified Expressions	4-99
4.8 Allocators	4-101
4.9 Static Expressions and Static Subtypes	4-120
4.10 Universal Expressions	4-128
5 Statements	5-1
5.1 Simple and Compound Statements	5-1
5.2 Assignment Statements	5-4
5.2.1 Array Assignments	5-9
5.3 If Statements	5-12
5.4 Case Statements	5-14
5.4.a Basic Case Statement Properties	5-14
5.4.b When others Can Be Omitted	5-20
5.5 Loop Statements	5-22
5.5.a Properties of All Loops	5-22
5.5.b FOR Loops	5-24
5.5.c WHILE Loops	5-28
5.5.d Continuous Loops	5-28
5.6 Blocks	5-29
5.7 Exit Statements	5-30
5.8 Return Statements	5-32
5.9 Goto Statements	5-34
6 Subprograms	6-1
6.1 Subprogram Declarations	6-1
6.2 Formal Parameters	6-6
6.3 Subprogram Bodies	6-11
6.3.1 Conformance Rules	6-12
6.3.2 Inline Expansion of Subprograms	6-16
6.4 Subprogram Calls	6-18
6.4.1 Parameter Associations	6-20
6.4.2 Default Parameters	6-35
6.5 Function Subprograms	6-36
6.6 Overloading of Subprograms	6-37
6.7 Overloading of Operators	6-40

Table of Contents

7 Packages	7-1
7.1 Package Structure	7-1
7.2 Package Specifications and Declarations	7-2
7.3 Package Bodies	7-4
7.4 Private Type and Deferred Constant Declarations	7-7
7.4.1 Private Types	7-9
7.4.2 Operations on Private Types	7-14
7.4.3 Deferred Constants	7-23
7.4.4 Limited Types	7-27
8 Visibility Rules	8-1
8.1 Declarative Regions	8-1
8.2 Scope of Declarations	8-1
8.3 Visibility of Identifiers	8-3
8.3.a Labels, Loop Names, and Block Names	8-18
8.3.b Loop Parameters	8-22
8.3.c Records	8-23
8.3.d Enumeration Literals	8-24
8.3.e Subprogram and Entry Parameters	8-25
8.3.f Packages	8-26
8.4 Use Clauses	8-27
8.5 Renaming Declarations	8-32
8.6 The Package Standard	8-40
8.7 The Context of Overload Resolution	8-44
8.7.a General Rules for Overloading Resolution	8-45
8.7.a.1 Nonoverloadable Constructs	8-46
8.7.a.2 Syntactic Ambiguity	8-47
8.7.a.3 Ambiguities Regarding Visibility	8-51
8.7.a.4 Complete Contexts for Overloading Resolution	8-52
8.7.a.5 A Model for an Overloading Resolution Algorithm	8-53
8.7.a.6 An Algorithm for Overloading Resolution	8-55
8.7.a.7 Implicit Conversions of Numeric Types	8-57
8.7.b Specific Overloading Resolution Rules	8-62
9 Tasks	9-1
9.1 Task Specifications and Task Bodies	9-1
9.2 Task Types and Task Objects	9-3
9.3 Task Execution -- Task Activation	9-6
9.4 Task Dependence -- Termination of Tasks	9-12
9.5 Entries, Entry Calls, and Accept Statements	9-18
9.6 Delay Statements, Duration, and Time	9-34
9.7 Select Statements	9-37
9.7.1 Selective Waits	9-37
9.7.2 Conditional Entry Calls	9-40
9.7.3 Timed Entry Calls	9-42

Table of Contents

9.8 Priorities	9-44
9.9 Task and Entry Attributes	9-46
9.10 Abort Statements	9-49
9.11 Shared Variables	9-51
10 Program Structure and Compilation Issues	10-1
10.1 Compilation Units -- Library Units	10-1
10.1.1 With Clauses	10-5
10.2 Subunits of Compilation Units	10-9
10.3 Order of Compilation	10-15
10.4 Program Library	10-24
10.5 Elaboration of Library Units	10-24
10.6 Program Optimization	10-29
11 Exceptions	11-1
11.1 Exception Declarations	11-1
11.2 Exception Handlers	11-3
11.3 Raise Statements	11-5
11.4 Exception Handling	11-6
11.5 Exceptions Raised During Task Communication	11-9
11.6 Exceptions and Optimizations	11-10
11.7 Suppressing Exceptions	11-12
12 Generic Units	12-1
12.1 Generic Declarations	12-1
12.1.1 Generic Formal Objects	12-10
12.1.2 Generic Formal Types	12-15
12.1.3 Generic Formal Subprograms	12-24
12.2 Generic Bodies	12-31
12.3 Generic Instantiation	12-32
12.3.1 Matching Rules for Formal Objects	12-39
12.3.2 Matching Rules for Formal Private Types	12-46
12.3.3 Matching Rules for Formal Scalar Types	12-53
12.3.4 Matching Rules for Formal Array Types	12-54
12.3.5 Matching Rules for Formal Access Types	12-58
12.3.6 Matching Rules for Formal Subprograms	12-63

Table of Contents

13 Representation Clauses and Implementation-Dependent Features	13-1
13.1 Representation Clauses	13-1
13.1.a Multiple Representations for a Type	13-6
13.1.b Forcing Occurrences	13-7
13.1.c Representation of Derived Types	13-12
13.1.d The pragma PACK	13-14
13.2 Length Clauses	13-16
13.2.a Size Specifications	13-17
13.2.b Collection Size Specifications	13-28
13.2.c Task Storage Size Specifications	13-31
13.2.d Small Specifications	13-33
13.3 Enumeration Representation Clauses	13-37
13.4 Record Representation Clauses	13-40
13.5 Address Clauses	13-47
13.5.1 Interrupts	13-55
13.6 Change of Representation	13-56
13.7 The Package System	13-57
13.7.1 System-Dependent Named Numbers	13-59
13.7.2 Representation Attributes	13-60
13.7.3 Representation Attributes of Real Types	13-62
13.8 Machine Code Insertions	13-63
13.9 Interface to Other Languages	13-65
13.10 Unchecked Programming	13-68
13.10.1 Unchecked Storage Deallocation	13-68
13.10.2 Unchecked Type Conversions	13-69
14 Input-Output	14-1
14.1 External Files and File Objects	14-1
14.2 Sequential and Direct Files	14-3
14.2.1 File Management	14-3
14.2.2 Sequential Input-Output	14-13
14.2.3 Specification of the Package Sequential_IO	14-19
14.2.4 Direct Input-Output	14-19
14.2.5 Specification of the Package Direct_IO	14-23
14.3 Text Input-Output	14-23
14.3.1 File Management	14-25
14.3.2 Default Input and Output Files	14-29
14.3.3 Specification of Line and Page Lengths	14-31
14.3.4 Operations on Columns, Lines, and Pages	14-32
14.3.5 GET and PUT Procedures	14-38
14.3.6 Input-Output of Characters and Strings	14-39
14.3.7 Input-Output for Integer Types	14-42
14.3.8 Input-Output for Real Types	14-51
14.3.9 Input-Output for Enumeration Types	14-62
14.3.10 Specification of the Package Text_IO	14-67
14.4 Exceptions in Input-Output	14-67
14.5 Specification of the Package IO_Exceptions	14-68
14.6 Low Level Input-Output	14-68

Chapter 2

Lexical Elements

2.1 Character Set

Semantic Ramifications

S1. Although RM 2.1/3 states "The basic character set is sufficient for writing any program," this does not mean an implementation is allowed to support only the basic character set. An implementation must support the entire set of characters defined in the syntactic category `graphic_character`. RM 2.1/3 means a *programmer* need not use the full character set in order to write an Ada program. In particular, the replacements allowed by RM 2.10 can be used by programmers if the codes for sharp (#), bar (|), and quotation (") do not have a suitable graphical representation.

S2. "Format effector" is a term defined by ISO¹ standard 646. The only nongraphic control characters allowed in an Ada program are the format effectors. The use of any other control characters, such as backspace, DEL, and NUL, is illegal.

Changes from July 1982

S3. There are no significant changes.

Changes from July 1980

S4. % is no longer a member of the basic character set.

S5. Each graphic character corresponds to a unique ISO standard 646 seven-bit coded character.

S6. Graphic characters may be represented by different graphical symbols in alternate national representations of the ISO character set.

Legality Rules

L1. Only graphic characters and the format effectors (horizontal tabulation, vertical tabulation, carriage return, line feed, and form feed) are allowed in Ada programs (RM 2.1/1).

Test Objectives and Design Guidelines

T1. Check that the basic character set is accepted outside of string literals and comments.

T2. Check that the lower case letters and other special characters are accepted (see IG 2.3/T1, IG 2.4/T2, IG 2.4.2/T7, IG 2.6/T4, and IG 2.7/T4).

T3. Check that the only control characters allowed in an Ada program are the format effectors (see IG 2.2/T5, IG 2.5/T2, IG 2.6/T5, and IG 2.7/T5).

¹ISO stands for International Organization for Standardization.

2.2 Lexical Elements, Separators, and Delimiters

Semantic Ramifications

S1. The character sequence '--' is not considered a compound delimiter (i.e., a lexical unit) since it introduces a comment, which is itself a lexical unit. Similarly, the quotation character (") is not considered a delimiter since it is part of a string literal. Finally, the sharp character (#) is not a delimiter, but is part of a based number.

S2. The end of a line need not be physically represented by character codes in the text. For example, lines could be represented physically as a string of graphic characters preceded by a character count.

S3. A period is not a delimiter when it appears in a real literal, but it is a delimiter when it appears in an expanded name. Similarly, a colon is not a delimiter when used within a based literal as a replacement for the sharp character (RM 2.10/3).

S4. Separators are not lexical elements. They may appear anywhere in a program as long as they are not included in a lexical element. Note that expanded names are made up of lexical elements. For example, P.F is composed of three lexical elements: P, ., and F. Separators may appear between these lexical elements.

S5. A lexical element is composed of the longest sequence of characters that satisfies the lexical rules, since RM 2.2/2 says in effect, that if a lexical element is preceded or followed by a character, it must not be possible to adjoin the character to the lexical element to produce a different lexical element.

S6. Lexical processing of the apostrophe is complicated by the three contexts in which it is used: character literals, attribute naming, and qualified expressions, e.g.:

```

if ('in'a'...) then
if ('=' ) then           -- two character literals, '(' and ')'
CHARACTER ('a')          -- 'a' qualified by CHARACTER
T'BASE'FIRST             -- attribute naming
POOL(K) 'TERMINATED      -- attribute naming

```

To resolve the potential lexical ambiguities implied here, note that the sequence "apostrophe, character, non-apostrophe" always means the first apostrophe is a delimiter (if it is already known not to be the terminating apostrophe for a character literal). If the sequence "apostrophe, character, apostrophe" is seen, and the preceding lexical element is an identifier, the sequence should be considered a character literal if the identifier is a reserved word. If it is not a reserved word, the first apostrophe should be considered a delimiter. In short, to perform lexical analysis with only two-character lookahead, you must sometimes know whether or not the preceding lexical element is a reserved word.

S7. Two-character lookahead is also needed to process based literals that use colons instead of sharps. Consider:

```

X : INTEGER range 0..2 := 1;
Y : INTEGER range 0..2:10 := 1;
Z : INTEGER range 0..2#10# := 1;

```

When a digit is followed by a colon, it is not possible to tell if the sequence is the start of a based number or if the colon is the beginning of the := compound delimiter. This ambiguity can only be resolved by looking at the character that follows the colon.

S8. The rule requiring that a separator appear between numeric literals and/or identifiers (RM 2.2/4) makes a construct such as:

if X=10mod-2then

illegal, since the literal 10 is not separated from the identifiers mod and the literal 2 is not separated from then.

Changes from July 1982

- S9. A program is now defined as one or more compilations (instead of as a single compilation).
- S10. The horizontal tabulation character is no longer defined as being equivalent to one or more spaces.
- S11. Separators are allowed at the beginning and the end of a compilation.

Changes from July 1980

- S12. Delete and null characters are no longer simply ignored; they are specifically illegal (RM 2.1/1).
- S13. Format effectors are no longer the only means of indicating the end of a line.
- S14. Horizontal tabulation is not a separator within a comment.

Legality Rules

- L1. A lexical element consists of the longest sequence of characters satisfying the definition of a lexical element (RM 2.2/2).
- L2. At least one separator must appear between adjacent numeric literals and/or identifiers (RM 2.2/4).
- L3. The only ASCII control characters permitted in Ada programs are CR, LF, VT, HT, and FF (RM 2.1/1).
- L4. No lexical element can extend over a line boundary (since a line boundary is a separator) (RM 2.2/3).

Test Objectives and Design Guidelines

- T1. Check that an identifier, reserved word, compound symbol, numeric literal, character literal, string literal, or comment cannot be continued across a line boundary.
Implementation Guideline: This test is primarily intended to check that implementations accepting only fixed-length input records do not catenate records, but instead treat end-of-record as a separator. The test should be coded to ensure that the attempt to continue a construct across a "line" boundary occurs at an input record boundary. Since input record lengths will vary from one implementation to the next, this test must be parameterized to accept different input record lengths. For implementations that accept variable-length input records up to a maximum length, the "continued" construct must appear at the end of a maximum-length input record.
- T2. Check that the delimiters are accepted, especially the compound symbols.
- T3. Check that at least one space must separate adjacent identifiers (including reserved words) and/or numbers.
- T4. Check that lexical units such as identifiers (including reserved words), numbers, and compound symbols cannot contain spaces.
- T5. Check that ASCII control characters other than carriage return, line feed, vertical tabulate, horizontal tabulate, and form feed, cannot appear in Ada programs outside of character literals, string literals, and comments.
Implementation Guideline: See IG 2.5/T2, IG 2.6/T5, and IG 2.7/T5 for tests of character literals, string literals, and comments, respectively.
Implementation Guideline: Each control character should appear in a separate test. The characters signifying end of line need not be checked here; they are checked in T1.

- T6. Check that any of the following — carriage return (CR), line feed (LF), vertical tab (VT), and form feed (FF) — are permitted outside string literals and are considered as signaling the end of a line.

Implementation Guideline: Include LF and a sequence of one or more CRs as line terminators, as well as more customary sequences such as CR LF and CR FF.

Check that horizontal tabulation is allowed in comments and between lexical units, where it is considered equivalent to a space.

Check that one or more spaces, horizontal tabulation characters, or end-of-line characters are permitted at the beginning or the end of a compilation unit.

Implementation Guideline: Write separate tests for end-of-line characters and the other separators.

Implementation Guideline: Some implementations may not be able to create or accept files that do not terminate with an end of line. In this case, the test using blanks and HTs at the end of the compilation file will not be applicable.

2.3 Identifiers

Semantic Ramifications

- S1. Identifiers (including those used to name library units or subunits) can be as long as the maximum line length allowed by an implementation. If library unit (and subunit) names can exceed the length allowed by a standard linker, an Ada implementation must provide a special linker.

Changes from July 1982

- S2. There are no changes.

Changes from July 1980

- S3. There are no significant changes.

Legality Rules

- L1. No identifier can be longer than the maximum input line length permitted by an implementation (RM 2.2/3).

Test Objectives and Design Guidelines

- T1. Check that upper and lower case letters are equivalent in identifiers (including reserved words). (See also 8.3.d/T1.)

Implementation Guideline: Try some all-upper, all-lower, and mixed case identifiers.

- T2. Check that consecutive, leading, and/or trailing underscores are not permitted in identifiers.

- T3. Check that identifiers can be as long as the maximum input line length permitted by the implementation, and that all characters are significant (e.g., not just the first 8 or 16, or not just the first m and last n characters). Try identifiers serving as variables, enumeration literals, subprogram names, parameter names, entry names, record component names, type names, package names (both library units and subunits), statement labels, block labels, loop labels, task names, and exception names.

Implementation Guideline: Maximum-length subprogram names and package names should be checked in separate tests.

- T4. Check that none of the characters ? \$ @ # ' is permitted in identifiers.

Check that none of the characters "&()*+,-./:;<=>|" is permitted in identifiers.

Implementation Guideline: Check specifically for:

```

A.B : INTEGER;
C*D : BOOLEAN;
type X&Y is ...;

```

- T5. Check that an identifier cannot start with a digit (see IG 2.2/T3).
- T6. Check that the underline character is significant in an identifier.

2.4 Numeric Literals

Semantic Ramifications

- S1. Any preceding + or - sign is not part of the numeric literal. Instead, the sign combined with the literal forms an arithmetic expression.
- S2. Real literals are often implicitly converted to some real type (RM 4.6/15). Such conversions must be exact if the real literal is a model number of the target type since conversion is a predefined operation giving a real result, and conversion of a *universal_real* value to a model number of another type must be done exactly (RM 4.5.7/6). If 'DIGITS for a floating point type is N, the exact decimal representation of 'LARGE will require more than N digits, and this decimal value must be converted exactly to 'LARGE. For example, if N = 6, 'LARGE is 19_342_803-_890_462_029_940_523_008.

Changes from July 1982

- S3. There are no significant changes.

Changes from July 1980

- S4. There are no significant changes.

Legality Rules

- L1. No numeric literal can contain more characters than the maximum input line length permitted by an implementation (RM 2.2/3).

Test Objectives and Design Guidelines

- T1. Check that underscores are not permitted to:
 - a. be consecutive within a numeric literal,
 - b. lead or trail in a numeric literal,
 - c. be adjacent to (on either side of) the '#', ':', '.', 'E', or 'e' characters in a numeric literal.

Implementation Guideline: Check for both based and decimal literals.
- T2. Check that 'E' and 'e' may be used in both decimal literals and based_numbers.
Check that numeric literals yield the correct values for both fixed and floating point types.
- T3. Check that leading zeros in numeric literals (and in integral subparts such as bases and exponents) are allowed.
Check that trailing zeros in the fractional part of real literals are allowed.
Check that numeric literals can be as long as the maximum input line length permitted by an implementation.
Implementation Guideline: Use leading or trailing zeroes to extend the length of the literal without producing a literal whose value is too large.

- T4. Check that negative exponent values are forbidden in integer literals and are allowed in real literals (see IG 2.4.1/T4 and IG 2.4.2/T5).
- T5. Check that leading/trailing decimal points are not permitted in real literals (including based literals).
- T6. Check that underscore characters in numeric literals have no effect on the represented value (see IG 2.4.1/T6 and IG 2.4.2/T2).
- T7. Check that real literals must contain a point.
- T9. Check that integer literals must not contain a point.

2.4.1 Decimal Literals

Semantic Ramifications

Changes from July 1982

- S1. There are no significant changes.

Changes from July 1980

- S2. A minus sign is explicitly forbidden in the exponent of an integer literal (i.e., 0E-0 is no longer allowed).

Legality Rules

- L1. No decimal literal can contain more characters than the maximum input line length permitted by an implementation (RM 2.2/3).
- L2. If a decimal literal without a decimal point has an exponent part, the exponent must not contain a minus sign (RM 2.4.1/4).

Test Objectives and Design Guidelines

- T1. Check that underscores are not permitted to:
 - a. be consecutive within a decimal literal,
 - b. lead or trail in a decimal literal,
 - c. be adjacent to (on either side of) the '.', 'E', or 'e' characters in a decimal literal (see IG 2.4/T1).
- T2. Check that 'E' and 'e' may be used in decimal literals (see IG 2.4/T2).
Check that decimal literals yield the correct values (see IG 2.4/T2).
- T3. Check that leading zeros in decimal literals are ignored (see IG 2.4/T3).
Check that trailing zeros in the fractional part of real literals are ignored (see IG 2.4/T3).
Check that decimal literals can be as long as the maximum input line length (see IG 2.4/T3).
- T4. Check that negative exponent values are forbidden in integer literals.
Check that negative exponent values are allowed in real literals (implicitly checked by IG 2.4.1/T13).
- T5. Check that leading/trailing points are not permitted in real literals (see IG 2.4/T5).

- T6. Check that underscore characters in decimal literals do not affect the represented value.
- T13. Check that decimal values of 'LARGE and 'SMALL are represented exactly (as long as the line length is sufficiently long).

Implementation Guideline: Use digit values of 5-29, in separate tests.

2.4.2 Based Literals

Semantic Ramifications

S1. The same literal value can be written in different ways in the source text of a program, e.g., 0.01E1 and 0.1; 2#0.01# and 0.25; and 16#0.F# and 4#0.33#. Real literals that do not represent model numbers need not have the same representation in an executed program. For example, 0.1 is not a floating point model number, so different occurrences of 0.1 need not have the same representation, and the representation for 0.1 need not be the same as that for 0.01E1. Of course, any representation of 0.1 must lie within the correct model interval for the floating point type (RM 4.5.7).

S2. The TEXT_IO input routines read real values according to the syntax of a real literal. Since the syntax of a based real literal allows any letter to occur between the sharp signs (#), this means the reading of a real literal cannot stop just because a letter greater than F is found while attempting to read a real literal. See IG 14.3.8/S for further discussion.

S3. RM 2.10/3 allows a based literal to be written with colons replacing each sharp. This affects lexical analysis. Two-character lookahead is needed to process based literals that use colons instead of sharps. Consider:

```
X : INTEGER range 0..2:= 1;
Y : INTEGER range 0..2:10:= 1;
Z : INTEGER range 0..2#10#:= 1;
```

When a digit is followed by a colon, it is not possible to tell if the sequence is the start of a based number or if the colon is the beginning of the := compound delimiter. This ambiguity can only be resolved by looking at the character that follows the colon.

Changes from July 1982

- S4. There are no significant changes.

Changes from July 1980

- S5. There are no significant changes.

Legality Rules

- L1. The base value must be greater than or equal to 2 and less than or equal to 16 (RM 2.4.2/1).
- L2. The letters and digits occurring between sharps must have a value less than the value of the base (RM 2.4.2/4).
- L3. The exponent for an integer based literal must not contain a minus sign (RM 2.4.1/4).
- L4. An integer based literal must not contain a point (RM 2.4/1).
- L5. If one sharp is replaced with a colon, the other must also be replaced (RM 2.10/3).
- L6. No based literal can contain more characters than the maximum input line length permitted by an implementation (RM 2.2/3).

Test Objectives and Design Guidelines

- T1. Check that a based literal always yields a nonnegative value.
- T2. Check that nonconsecutive embedded underscores are permitted in every part of a based literal, and do not affect the value.
- T3. Check that based literals with bases 2 through 16 all yield correct values.
- T4. Check that the digits and extended digits of a based literal are within the correct range for the number's base.
- T5. Check that negative exponents are forbidden in integer based literals.
Implementation Guideline: Use an exponent of -0 in one case.
Check that negative exponents are allowed in real based literals (implicitly checked by IG 2.4.1/T13).
- T6. Check that the base must not be less than 2 or greater than 16.
- T7. Check that letters in a based literal may appear in upper or lower case.
- T8. Check that underscores may not be adjacent to the '#' (see IG 2.4/T1).
- T9. Check that 'E' and 'e' may appear in based literals (see IG 2.4/T2).
- T10. Check that leading zeros in based literals are ignored (see IG 2.4/T3).
Check that based literals can be as long as the maximum input line length (see IG 2.4/T3).
- T11. Check that two-character lookahead is used for based literals having colons in place of sharps.

2.5 Character Literals**Semantic Ramifications**

- S1. The type of a character literal is determined contextually (see IG 8.7). The determination uses the rules for resolving the type of a subprogram identifier, since a character literal is considered to be declared as a function (RM 3.5.1/3).
- S2. Character literals can be overloaded (RM 8.7/1).

```

package P is
  type ENUM is ('A', 'B'); -- overloads 'A' in CHARACTER
  V_ENUM : ENUM := 'A';   -- legal
  W_ENUM : ENUM := 'C';   -- illegal; 'C' does not have type ENUM
end P;

```

- S3. Of course, the character literals must be visible.

```

with P;
package Q is
  V : P.ENUM := 'A';      -- illegal; P.'A' not visible
end Q;

```

- S4. The type of a character literal is determined in part by the visibility of declared character literals:

```

with P; use P;
package R is

```

```

      B : BOOLEAN := 'A' = 'D'; -- legal
      C : BOOLEAN := 'A' = 'A'; -- illegal
end R;

```

Since the only visible 'D' is STANDARD.'D' (RM C/13), 'A' and = are uniquely resolvable in the first case. 'A' = 'A' is illegal because the operand can have either type ENUM or type CHARACTER, and the equality operators for ENUM and CHARACTER values are both visible.

S5. It is not always possible to replace a character string with a catenation of character strings. For example, although a multidimensional array aggregate can be written using a string literal, e.g., (1..2 => "EF"), it is not legal to replace the literal with a catenation of literals, e.g., (1..2 => "E" & "F") (see IG 4.3.2/S).

Changes from July 1982

S6. There are no significant changes.

Changes from July 1980

S7. There are no significant changes.

Legality Rules

- L1. A character literal cannot extend across a line boundary (RM 2.2/3).
- L2. The character enclosed in apostrophes must be one of the 95 ASCII graphic characters, including space (RM 2.5/1).
- L3. The type of a character literal is determined by the visibility of corresponding character literal declarations, and by the context in which the literal is used (RM 8.7/1-2).

Test Objectives and Design Guidelines

- T1. Check that all character literals can be written.
Implementation Guideline: Check the position number within CHARACTER.
 Check that upper and lower case characters are distinct.
- T2. Check that nongraphic characters are not allowed in character literals, including format effectors.
Implementation Guideline: Check all nongraphic characters except those signifying end of line (see IG 2.2/T1 for end-of-line test).
- T3. Check that forms such as if'b'ln'a'..'c' and T'BASE'FIRST are lexically analyzed correctly.
- T4. Check that the type of an overloaded character literal is correctly determined, using the visibility of enumeration types and operations on these types.
Implementation Guideline: Use a case like 'A' = 'D', which is legal, and a case like 'A' = 'A' which is not legal.

2.6 String Literals

Semantic Ramifications

- S1. The lower bound of a string literal is determined according to the rules for positional aggregates (RM 4.2/3; see also IG 4.2/S).
- S2. Although the note in RM 2.6/6 suggests that it is always possible to substitute a catenation of string literals for a single string literal, in fact, there are several contexts in which this substitution is not allowed, either because the bounds of the catenation will not be the same as

the bounds implied for the literal, or because the rules of the language require that a string literal be used, not an expression yielding a string value. For example, although the value of concatenated string literals is equivalent to the value of a string literal, the bounds of the concatenation need not be the same as the bounds that would be associated with the literal:

```
subtype STR is STRING (2..4);
procedure P(X : STR (2..4));
```

The call P("ABC") is legal since the string literal is given the bounds of the formal parameter. However, the call P("A" & "BC") is illegal since the lower bound of this concatenation is the lower bound of the left operand, "A" (RM 4.5.3/4), and the lower bound of "A" is the lower bound of its index subtype (RM 4.2/3 and RM 4.3.2/9), i.e., 1. Hence, "A" & "BC" has the bounds 1..3, and these bounds do not satisfy (i.e., equal) the bounds of the formal parameter.

S3. The use of a string literal can be legal where a concatenation would be illegal. In particular, string literals can be used when writing aggregates of a multidimensional array type, but a concatenation of string literals cannot be used in place of the string literal:

```
type MULTI is array (1..3, 1..2) of CHARACTER;
VAR1 : MULTI := (1..3 => "AB");
VAR2 : MULTI := (1..3 => ('A', 'B'));
VAR3 : MULTI := (1..3 => "A" & "B"); -- illegal
```

The concatenation is illegal because RM 4.3.2/2 only allows string literals (or aggregates) to be used to give the value of the last dimension of a multidimensional array aggregate.

S4. String literals are never padded in assignment or comparison contexts.

S5. The character literals corresponding to the graphic characters contained within a string literal must be visible at the place of the string literal (RM 4.2/5). Moreover, since a string literal creates an array of values whose components must have a particular type, the characters used within the literal must belong to the corresponding character type. For example:

```
package P is
  type NS is array (1..5) of CHARACTER;
end P;

with P;
package Q is
  VAR : P.NS := "ABCDE"; -- legal
end Q;
```

The string literal is legal because the character literals are of type CHARACTER and are directly visible within Q. In addition, string literal formation and assignment are basic operations (RM 3.3.3/4,7) declared in P, and are visible throughout P's scope (RM 8.3/18). Since P's scope includes Q, the use of these operations is legal within Q. But suppose the component type of the array is declared directly within a package:

```
package R is
  type ENUM is ('A', 'B', 'C', BLUE);
  type STR is array (1..3) of ENUM;
end R;

with R;
package S is
  VAR : R.STR := "ABC"; -- illegal
end S;
```

The string literal is illegal because the character literals for type ENUM are not directly visible. If we added a use clause to package S so the enumeration literals were directly visible, then "ABC" would be a legal literal of type R.STR, but "ABD" would not be legal because the only visible 'D' is a value of type CHARACTER, and the string literal requires values of type ENUM.

S6. Evaluation of a string literal can raise CONSTRAINT_ERROR if the characters used in the string literal do not satisfy the constraint of the literal's component type:

```
type NO_CHAR is new CHARACTER range ASCII.NUL .. ASCII.BEL;
                                -- no graphic characters
type NO_LIT is array (POSITIVE range <>) of NO_CHAR;
x : NO_LIT (1..3) := "ABC";    -- CONSTRAINT_ERROR
```

"ABC" raises CONSTRAINT_ERROR because none of the values 'A', 'B', or 'C' belong to the NO_LIT component subtype. (Since NO_CHAR is derived from CHARACTER, NO_CHAR is a character type and all the character literals in CHARACTER are derived for NO_CHAR's base type. Hence, 'A', 'B', and 'C' are values of NO_CHAR's base type, and they are visible, so "ABC" is a *legal* string literal.)

S7. See IG 4.2/S for a discussion of how the type of a string literal can be determined. (In particular, the characters used within a string literal do not help determine its type.)

S8. A string literal can also be written using % signs as string brackets instead of quotation characters (RM 2.10/4). When enclosing % signs are used, the sequence of characters must not contain a quotation character, and any % sign in the string's value must be doubled in the enclosed sequence of characters.

Changes from July 1982

S9. There are no significant changes.

Changes from July 1980

S10. There are no significant changes.

Legality Rules

- L1. A string literal cannot extend across a line boundary (RM 2.2/3).
- L2. Nongraphic ASCII characters are not permitted in string literals (RM 2.2/2).
- L3. If a percent sign occurs at the beginning of a string literal (replacing a quotation character), the final quotation character must also be replaced, the enclosed sequence of characters must not contain any quotation characters, and any percent characters within the original sequence of characters must be doubled (RM 2.10/4).
- L4. The type of a string literal must be determinable solely from the context in which the literal appears, but with the understanding that the literal is a value of a one-dimensional array type whose component type is a character type (RM 4.2/4).
- L5. The character literals corresponding to the graphic characters contained within a string literal must be visible at the place where the string literal appears (RM 4.2/5).

Exception Conditions

- E1. CONSTRAINT_ERROR is raised for a null string literal if the lower bound is 'FIRST' of the index *base* type (RM 4.2/3).
- E2. CONSTRAINT_ERROR is raised for a string literal if any character in the literal does not belong to the component subtype (RM 4.2/3 and RM 4.3.2/11).

Test Objectives and Design Guidelines

- T1. Check that string literals started with a quotation character cannot end with a percent character, and vice versa.
- T2. Check that `"` must be doubled when used within string literals.
- T3. Check that string literals cannot cross line boundaries (see IG 2.2/T1).
- T4. Check that all printable characters are permitted in string literals.
Implementation Guideline: Check that the replacement characters (see RM 2.10) are represented correctly in a string literal, e.g., `'\'` is represented as `'\'`, not as `ASCII.BAR`.
- T5. Check that nongraphic characters cannot be included in string literals.
- T6. Check that all ASCII characters, including control characters, can appear anywhere in string values.
Implementation Guideline: In particular, check that string values can include NUL and DEL.
- T7. Check that a string literal having the maximum permitted line length can be generated.
- T8. Check that upper and lower case letters are distinct within string literals.
- T9. Check that string literals have the appropriate bounds (see IG 4.3.2/T14).

2.7 Comments**Semantic Ramifications**

S1. Although the horizontal tabulation character is allowed in a comment, no other control characters are allowed (RM 2.2/2). However, AI-00339 allows (but does not require) an implementation to accept an extended character set as long as the additional characters only appear in comments. (This allows understandable comments to be written in languages other than English, e.g., French and Japanese.)

Approved Interpretations

S2. An implementation is allowed to accept an extended character set (i.e., graphic characters whose codes do not belong to the ISO seven-bit coded character set (ISO standard 646) as long as the additional characters appear only in comments (i.e., the additional characters only appear after two adjacent hyphens and precede the end of the line) (AI-00339).

Changes from July 1982

S3. There are no significant changes.

Changes from July 1980

S4. There are no significant changes.

Test Objectives and Design Guidelines

- T1. Check that a comment is terminated by the end of the line, i.e., not by the next `--`, whether on the same line or on the next line.
- T2. Check that legal Ada statements and pragmas contained in comments have no effects when contained in comments.
- T3. Check that consecutive hyphens in a string literal are not treated as the beginning of a comment.

Check that a single double quote can appear in a comment.

T4. Check that every graphic character can appear in a comment.

Check that horizontal tabulation characters can appear in a comment (see IG 2.2/T6).

T5. Check that nongraphic characters (other than horizontal tabulation and end-of-line characters) cannot appear in a comment.

2.8 Pragmas

Semantic Ramifications

S1. A pragma may follow any semicolon except one appearing in a discriminant part or a formal part. Syntactically, a formal_part declares the formal parameters of an entry, an accept statement, or a subprogram; therefore, forbidding pragmas in formal parts means pragmas are not allowed after the declaration of such formal parameters. Pragmas, however, can appear after generic formal parameter declarations, since these declarations do not, syntactically, constitute a formal_part. Since pragmas end with a semicolon, one pragma can follow another.

S2. Pragmas are allowed "at any place where the syntax rules allow a construct defined by a syntactic category whose name ends with 'declaration,' 'statement,' 'clause,' or 'alternative,' or one of the syntactic categories, variant and exception_handler (RM 2.8/5). This means a pragma can precede any construct terminated by a semicolon. In particular, pragmas can appear at the beginning of a component list, at the beginning of a declarative part or sequence of statements, before the first declarative item in the visible or private part of a package, before the first declaration in a task specification, before a select alternative, before an entry call in a conditional or timed entry call, and before the alignment clause in a record type representation. In addition, pragmas can appear after comments if they could appear when the comment is not present (i.e., comments are ignored when determining the legality of a pragma's position).

S3. Pragmas are not allowed "in place of" a construct. For example, a sequence of statements must contain at least one statement; a pragma cannot be given in place of the required statement.

S4. Pragmas are also allowed "at any place where a compilation unit would be allowed," i.e., a sequence of "compilation units" presented to a compiler can consist of one or more pragmas, and a sequence of pragmas can appear before any compilation unit. Although pragmas are allowed before and after compilation units, there is no ambiguity in compilations containing more than one unit, since the pragmas that can occur before compilation units are different from those that can appear after.

S5. If a compilation unit contains a "pragma" that does not appear in one of these allowed places, the program is illegal, e.g., the occurrence of a LIST pragma in the middle of a statement or after a semicolon in a formal part is illegal.

S6. The position of most pragmas is further restricted by the RM in terms of the syntactic context in which they can appear, e.g., an INLINE pragma is only permitted in a declarative part or after a library unit in a compilation. If a pragma does not appear in its required place, it "has no effect" (RM 2.8/9). For example, placing an INLINE pragma in the statement list of a library subprogram would have no effect on the compilation unit. In particular, the unit would not be considered illegal; the pragma appears in a place allowed for pragmas in general. It just does not appear in a place allowed for the INLINE pragma.

S7. Similarly, an invalid pragma argument has no effect on the compilation unit; such pragmas are ignored. For example, a PRIORITY pragma with an out-of-range value or a nonstatic argument is not illegal, but is merely ignored. Also, an INLINE pragma whose argument is not

declared previously in the same declarative part (see RM 6.3.2/2) is ignored, but is not illegal. Finally, names appearing in arguments need not be visible according to the usual rules. In short, a syntactically well-formed but otherwise improper argument is never illegal. For example, the pragma `INTERFACE` requires a language *name*. Note that `PL/I` is not a name according to the Ada syntax. Therefore,

```
pragma INTERFACE (PL/I, SIN);      -- ignored
```

must be ignored; `PL/I` can only be considered an expression. On the other hand,

```
pragma INTERFACE (APL\360, SIN);   -- illegal
```

is illegal, because the back-slash character is allowed only in string literals, character literals, and comments.

S8. Pragma names are not reserved. In addition, the normal visibility rules do not apply to them (RM 8.3/1), so they never become hidden or overloaded as a result of a declaration. For example, a declaration such as

```
procedure LIST (I : INTEGER);
```

has no effect on the visibility of the `LIST` pragma.

S9. Although RM 2.8/8 states that predefined pragmas must be supported (i.e., recognized) by every implementation, this does not mean an implementation must obey the pragma. Some pragmas give "permission" to the implementation to perform some action (e.g., pragma `SUPPRESS`; RM 11.7/3,4). The implementation does not have to take advantage of such permission.

S10. A compiler that does not recognize a pragma will be unable to determine the type or value of an expression used in an argument if overloaded identifiers occur in the argument; in fact, the compiler will be unable to determine whether a name should be parsed as a name or as an expression. The compiler must verify that the argument is syntactically well-formed (or else it would not be able to determine where the pragma ends). But the compiler is not allowed to reject a program if the argument contains identifiers that cannot be associated with any visible declaration, or contains overloaded identifiers that cannot be resolved uniquely. In short, if an argument is invalid for any of these reasons, the pragmas "have no effect" on the rest of the program; in particular, the program cannot be rejected. For example,

```
VAR : STRING(1..5);
pragma WHO_KNOWS (VAR (1, 3));
```

The pragma (and containing compilation unit) is not illegal, even though `VAR` seems to have too many subscripts. The "`VAR`" that appears in this implementation-defined pragma need not have anything to do with the visible declaration of `VAR`.

S11. The RM does not state whether listing is initially enabled or disabled. The choice is implementation dependent, e.g., under the control of a compiler option.

S12. To deduce the allowed position of each pragma (i.e., the position in which the pragma is allowed to have an effect), it is first necessary that the pragma appear in a legal position, i.e.,

- after a semicolon delimiter except within a formal part or discriminant part; or
- at any place where the syntax rules allow constructs whose names end with "declaration", "statement", "clause", or "alternative"; or
- at any place where the syntax rules allow the category, variant or exception-handler; or

- at any place where a compilation unit is allowed;

S13. Given that the pragma appears in a legal position for it to have an effect, any additional restrictions listed in Appendix B or stated in the body of the RM must also be obeyed, as specified for each pragma:

- LIST and PAGE may appear wherever a pragma may appear (RM B/6 and RM B/10).
- MEMORY_SIZE, STORAGE_UNIT, and SYSTEM_NAME may only appear at the start of a compilation before the first compilation unit (RM B/7, RM B/13, and RM B/15).
- OPTIMIZE may only appear in a declarative part (in particular, not in a package specification) (RM B/8).
- PRIORITY may only appear in a task specification or immediately within the outermost declarative part of the main program (RM B/11).
- CONTROLLED and SHARED must appear immediately within the declarative part or package specification in which the argument is declared, after the declaration of the argument (RM B/2 and RM B/12).
- INTERFACE must appear immediately within the declarative part or package specification in which its second argument is declared, after the argument's declaration, or it must appear after the declaration of a subprogram library unit and before any subsequent compilation unit (the second argument must be the name of the library unit) (RM B/5); see also IG 13.9/S.
- INLINE must appear immediately within a declarative part or package specification and after the declaration of each subprogram or generic subprogram named in the pragma, or after a subprogram or generic subprogram library unit in a compilation and before any subsequent compilation units (the argument must be the name of the library unit) (RM B/4, RM 6.3.2/2-3, and IG 6.3.2/S). (Note: the argument can be the name of a subprogram declared by generic instantiation.)
- ELABORATE must appear immediately following the context clause of a compilation unit; each argument must be the simple name of a library unit mentioned by the context clause (RM B/3).
- PACK must appear in the same context that a representation clause for its argument would appear (RM B/9).
- SUPPRESS must appear immediately within a declarative part or package specification, and must name a check identifier and the name of an object, type, subtype, subprogram, task unit, or generic unit declared earlier in the declarative part or package specification (RM B/14).

S14. Although new implementation-defined pragmas are allowed, the RM does not allow extensions of language-defined pragmas, e.g., by adding new arguments or new rules governing their use.

S15. Implementation-defined pragmas must be such that deletion of the pragma never changes the legality of the program. For example, an implementation-defined pragma must not have the effect of changing the character set, of enabling the subsequent inclusion of assembly language code, or of enabling language extensions.

S16. Expressions in pragmas never contain occurrences that force the default representation of a type to be determined. For further discussion, see IG 13.1.b/S.

S17. Although implementation-defined pragmas must not affect the legality of programs, they can increase the set of erroneous programs. For example, consider a pragma that says a subprogram will not be called recursively:

pragma NONRECURSIVE (PROC);

This pragma is equivalent to an assertion that the named subprogram will never be called recursively. An implementation can generate code that takes advantage of this information. Such code will have unpredictable results if the subprogram is, in fact, called recursively. This means the set of erroneous Ada programs will have been enlarged. However, since the presence or absence of the pragma does not influence the legality of the subprogram body or its callers, the pragma is allowed. Similar reasoning would allow a pragma ASSERT, whose argument is evaluated and, if false, raises an exception.

Changes from July 1982

S18. Pragmas may appear where the syntax allows declarations, statements, clauses, or alternatives, but not in place of these constructs.

S19. It is now stated explicitly that pragmas are allowed as compilation units.

S20. The pragmas defined in Annex B must be supported by every implementation.

S21. Pragmas containing names of compilation units need not appear after the named unit.

Changes from July 1980

S22. The expression given as the argument to a pragma need not be static.

S23. Implementation-defined pragmas do not affect the legality of a program, whether recognized or not.

S24. INCLUDE is no longer a predefined pragma.

S25. ELABORATE, PAGE, and SHARED are now predefined pragmas.

S26. The pragma SYSTEM is now called SYSTEM_NAME.

Legality Rules

L1. A pragma must only appear as a compilation unit, or before or after any syntactic construct terminated (as opposed to separated) by a semicolon (RM 2.8/3-5).

Test Objectives and Design Guidelines

Implementation Guideline: Several tests are designed to check the treatment of pragmas whose identifier is not recognized by the compiler. A highly non-mnemonic name should be used.

T1. Check that no pragma is allowed in the following contexts.

Implementation Guideline: Check LIST and an unrecognized pragma.

- a. after the reserved word **exception** in a block, subprogram body, package body, or task body;
- b. in an expression, statement, or actual parameter part;
- c. in a discriminant part or formal part;
- d. as an alignment clause in a record type representation;
- e. as the only statement in a sequence of statements, declarations, clauses, or alternatives.

- T2. Check that a predefined or an unrecognized pragma may have arguments involving identifiers that are not visible, or overloaded identifiers without enough contextual information to resolve the overloading, or visible identifiers used incorrectly (e.g., wrong types, incorrect number of subscripts, procedure call, operator symbol, etc.), or an arbitrary number of arguments, optionally in named notation.

Implementation Guideline: For each of the pragmas having an argument, try an unexpected form of argument.

- T3. Check that the argument to a predefined or unknown pragma cannot be a declaration, subtype indication, or assignment statement.

Implementation Guideline: Leave off the terminating semicolon.

- T4. Check that an entity with the name of a predefined pragma can be declared and does not hide the pragma.

Implementation Guideline: The following names should be used: CONTROLLED, ELABORATE, INLINE, INTERFACE, LIST, MEMORY_SIZE, OPTIMIZE, PACK, PAGE, PRIORITY, SHARED, STORAGE_UNIT, SUPPRESS, and SYSTEM_NAME).

Check that an unrecognized pragma with the same identifier as a declared entity or with a reserved identifier (check both cases) does not make the program illegal.

- T5. Check that LIST and PAGE work correctly (check both ON and OFF).

Implementation Guideline: Check that when LIST or PAGE are used between compilation units or at the beginning or end of a compilation, there is no problem.

Implementation Guideline: Check the use of LIST(OFF) at the beginning of a private part.

Implementation Guideline: Check LIST(OFF) and PAGE in the middle of a line.

Implementation Guideline: Check that these pragmas can appear at the beginning of a sequence of statements, declarations, alternatives, and clauses.

Implementation Guideline: Include a pragma that occurs after a comment.

- T6. For each implementation-defined pragma and an unrecognized pragma, check that:

- the arguments must be names or expressions, and
- deletion of the pragma from a program does not make the program illegal.

2.9 Reserved Words

Semantic Ramifications

S1. DIGITS, DELTA, and RANGE are reserved words used both in declarations and as predefined attributes (RM 3.5.8/4, RM 3.5.10/4 and RM 3.6.2/7). The lack of boldface type for these attributes does not imply they are unreserved.

Changes from July 1982

S2. There are no changes.

Changes from July 1980

S3. "abs" is now a reserved word.

Legality Rules

L1. Reserved words cannot be given user-defined meanings via declarations (RM 2.9/3).

Test Objectives and Design Guidelines

T1. Check that all the reserved words are actually reserved.

T2. Check that only the specified set of reserved words is actually reserved.

Implementation Guideline: Consider reserved words and keywords in languages other than Ada, e.g., PL/I (IBM F and optimizing versions, MULTICS, standard, PL/C, etc.), Pascal, JOVIAL (J73, J73I, J73C, J3, J3B), Tacpol, CMS-2, SPL/I, COBOL, FORTRAN 77, Algol 60, Algol 68. Check that these are not reserved in Ada unless they are the same as an Ada reserved word.

- T3. Check that all predefined attributes (except DIGITS, DELTA, and RANGE) and all predefined type and package names are not reserved.

2.10 Allowable Replacements of Characters

Semantic Ramifications

S1. The allowable replacement characters are not an option to the implementation, but an option to the writer of Ada programs. Each implementation must support these replacements.

S2. In a based number and in a string literal, the replacement characters must replace both the # or * at the beginning and the end of the literal. This is not the case when | replaces |. The following is legal.

```
type ENUM is (E1, E2, E3, E4);
E : ENUM;

case E is
  when E1 | E2 | E3 =>
    DO_SOMETHING;
  when E4 =>
    null;
end case;
```

S3. In addition, when checking whether discriminant parts, subprogram specifications, etc., conform (see RM 6.3.1), the use of the replacement characters does not affect conformance since the replacements do not change the meaning of a program (RM 2.10/5 and AI-00350). For example, the following formal parts conform to each other:

```
procedure P (S : STRING := (1 | 2 | 3 => 'x')); -- P1
-- Vertical bars in aggregate

procedure P (S : STRING := (1 ! 2 ! 3 => 'x')); -- conforms to P1
-- Exclamation points in aggregate

procedure P (S : STRING := (1 | 2 ! 3 => 'x')); -- conforms to P1
-- One of each in aggregate

procedure Q (S : STRING := "Foo"); -- Q1
-- Double quotes in string

procedure Q (S : STRING := %Foo%); -- conforms to Q1
-- Percent signs in string
```

S4. The replacement of # with a : in a based literal affects the lexical processing of a program because two-character lookahead is needed to see if a colon is part of a based number or part of an assignment compound delimiter:

```
X : INTEGER range 0..2:= 1;
Y : INTEGER range 0..2:10:= 1;
Z : INTEGER range 0..2#10#:= 1;
```

Changes from July 1982

S5. There are no significant changes.

Changes from July 1980

S6. There are no significant changes.

Legality Rules

- L1. The vertical bar can only be replaced by an exclamation mark if the vertical bar is being used as a delimiter (RM 2.10/2).
- L2. Both # characters in a based literal must be replaced by : when replacement characters are used (RM 2.10/3).
- L3. Both " characters for a string literal must be replaced by % if replacement characters are used (RM 2.10/4).
- L4. A string literal bracketed with percent signs cannot contain a quotation character (RM 2.10/4).

Test Objectives and Design Guidelines

- T1. Using colons instead of sharps, check that nonconsecutive embedded underscores are permitted in every part of a based literal, and do not affect the value.
- T2. Using colons instead of sharps, check that based literals with bases 2 through 16 all yield correct values.
- T3. Using colons instead of sharps, check that the digits and extended digits of a based literal are within the correct range for the number's base.
- T4. Using colons instead of sharps, check that negative exponents are forbidden in integer based literals.
Implementation Guideline: Use an exponent of -0 in one case.
- T5. Using colons instead of sharps, check that the base must not be less than 2 or greater than 16.
- T6. Using colons instead of sharps, check that letters in a based literal may appear in upper or lower case.
- T7. Using colons instead of sharps, check that underscores may not be adjacent to the colons.
- T8. Using colons instead of sharps, check that 'E' and 'e' may appear in based literals.
- T9. Using colons instead of sharps, check that leading zeros in based literals are ignored.
Using colons instead of sharps, check that based literals can be as long as the maximum input line length.
- T10. Check that a based literal cannot start with a # and end with a : and vice versa.
- T21. Check that a string literal delimited by a % character must not contain a " character.
Check that a % character must be doubled to appear in a string literal delimited by a % character.
- T31. Check that where | is used as a separator, I can be used instead.
Implementation Guideline: Check all contexts: choices in a case statement alternative or variant part, component associations of an aggregate, discriminant associations of a discriminant constraint, and in an exception handler.

Chapter 3

Declarations and Types

3.1 Declarations

Semantic Ramifications

- S1. The concept of an entity is important in defining the semantics of renaming declarations, which declare a new name for an entity but not a new entity (see IG 8.5/S).
- S2. A named number is an entity because it is declared by an object declaration.
- S3. The term "object" is further defined in RM 3.2/1-7.

Changes from July 1982

- S4. A generic instantiation is no longer regarded as generating implicit declarations.
- S5. Elaboration also applies to compilation units.

Changes from July 1980

- S6. The list of Ada entities has been extended by adding generic units and entry families.
- S7. The concept of elaboration is now strictly a run-time concept. The legality of a declaration is now clearly independent of whether or not it is ever elaborated.

3.2 Objects and Named Numbers

Semantic Ramifications

- S1. A function call does not return an object since the result of a function call is not listed in RM 3.2/2-7. A function call returns a value (RM 6.5/1).
- S2. Although the syntax allows the declaration of an object with an unconstrained array type (or an unconstrained record type with discriminants that have no defaults), the rules in RM 3.6.1/6 and RM 3.7.2/8 forbid such forms of object declaration for variables. These rules, however, do allow the declaration of a constant whose subtype indication denotes an unconstrained array or record type. In this case, the subtype of the constant is determined by the initial value (RM 3.6.1/7 and RM 3.7.2/9). For example:

```
X : constant STRING := "ABC";
```

X has bounds 1..3.

- S3. The := in object declarations represents the assignment operation since RM 3.2.1/8 specifies that initial values "are assigned" to objects. On the other hand, the := that specifies the default initial value for a subprogram in parameter does not represent the assignment operation. It is a notation used to specify a default actual parameter value, which is why in parameters having limited types can have default expressions (see IG 6.4.2/S).
- S4. Since assignment is not declared as an operation for a limited type (RM 7.4.4/1), no initial value can be provided for a limited type in an object declaration. Since constant declarations (except for object declarations having the form of a deferred constant declaration; RM 7.4/2) require an initialization expression, the lack of assignment means a constant object_declaration can never be written for a limited type.

S5. The effect of evaluating functions appearing in multiple declarations is supposed to be the same as the effect of the equivalent sequence of single declarations (RM 3.2/10). For example, suppose *F* is a function that returns successive integer values, starting with the value one. Now consider the effect of the following declaration:

```
type ARR is array (NATURAL range <>) of INTEGER;
S1, S2 : ARR (1..F) := (others => F);
```

The multiple object declaration must be considered equivalent to:

```
S1 : ARR (1..F) := (others => F);
S2 : ARR (1..F) := (others => F);
```

Since the subtype indication of an object declaration is evaluated before the initialization expression (RM 3.2.1/5-6 and RM 3.2.1/15), these declarations are, in turn, equivalent to:

```
S1 : ARR (1..F) := (1..1 => F);
S2 : ARR (1..F) := (F, F, F);
```

S1 has bounds 1..1 and initial value (1..1 => 2). S2 has bounds 1..3, and its initial value requires that *F* be evaluated three times. Since the order of evaluation of the expressions in an aggregate is not defined (RM 4.3.2/10), S2 can have any of the following values: (4, 5, 6), (5, 4, 6), (4, 6, 5), (6, 4, 5), (6, 5, 4), and (5, 6, 4). No other set of values and subtypes is allowed for S1 and S2.

S6. Similarly, consider a record declaration with default discriminants defined with the same function *F*:

```
type REC (D1, D2 : INTEGER := F) is
  record null; end record;
R1, R2 : REC;
```

Here *F* must be evaluated once for each default discriminant needed in an object declaration, and since there are, in effect, two object declarations, *F* must be evaluated four times. Since the order of evaluation of default expressions is not defined (RM 3.2.1/15), all that can be said about R1 is that its initial discriminant values are ... is not defined whether R1.D1 equals 1 or 2. R2's discriminants have the values 3 and 4.

S7. Now consider the declaration:

```
S3, S4 : array (1..F) of ARR (1..F) := (others => (others => F));
```

The *F*s in the constrained array definition are evaluated (in an undefined order); therefore, S3 either has the bounds 1..1 or 1..2, and similarly, its component type is either ARR (1..2) or ARR (1..1). In either case, its initial values are 3 and 4. S4 has bounds 1..5 or 1..6.

S8. Since multiple object declarations are considered different declarations, the use of a constrained array definition in a multiple object declaration declares objects having different types:

```
E, F : array (1..10) of INTEGER;
```

E and *F* cannot be assigned to each other or compared for equality because they have different (anonymous) base types.

Changes from July 1982

S9. There are no significant changes.

Changes from July 1980

S10. The semantics of multiple object declarations is different — the subtype indication or array type definition is evaluated once for each named object, and so is any initialization expression.

S11. An unconstrained array definition is not allowed in a constant declaration.

Test Objectives and Design Guidelines

- T1. Check that in a multiple object declaration, the subtype indication and the initialization expression are evaluated once for each named object that is declared, and the subtype indication is evaluated first. Check that the evaluations yield a result equivalent to the corresponding sequence of single object declarations.

Implementation Guideline: Check for all types: enumeration, integer, float, fixed, array, record, access, and private. Use constraints containing function calls to ensure that the subtype indication and initial value are evaluated once for each object, and that the subtype indication is evaluated first.

Implementation Guideline: Check for both variable and constant declarations.

Implementation Guideline: Include object declarations that use subtype indications and declarations that use a constrained array definition. Use at least one constrained array definition that has a component type with an index or discriminant constraint, to ensure that these constraints are also evaluated at the correct time.

Check that default discriminant expressions are evaluated once for each declared object in a multiple object declaration.

Implementation Guideline: Repeat this check for a generic unit when the multiple object declaration has a generic formal type and the actual parameter is a type with default discriminants.

- T2. Check that if a multiple object declaration (for a variable or a constant) uses a constrained array definition, the declared objects have different types (see IG 3.2.1/T6).
- T3. Check that when several record components are declared in a single component declaration, the subtype indication is evaluated once for each declared component, and any initialization expression is evaluated once for each component when an object of the type is declared (without an explicit initialization) (see IG 3.7/T3).
- T4. Check that when the full declaration of several deferred constants is given as a multiple declaration, the initialization expression is evaluated once for each deferred constant (see IG 7.4.3/T2).

3.2.1 Object Declarations**Semantic Ramifications**

S1. Although the value of a loop parameter can change, it is defined to be a constant by RM 3.2.1/2, so it cannot be used as the target of an assignment or passed as an in out or out parameter.

S2. The name declared by a renaming declaration is not included in the list of declared constant objects because a renaming declaration does not declare a new object; it only declares a new name for an entity (RM 8.5/1). If the renamed entity is a constant, then the new name denotes a constant (RM 8.5/4) and so cannot be used in contexts where a variable is required (e.g., on the left side of an assignment statement).

S3. If a constant has an access type, the designated object is nonetheless considered a variable. It is only the access value that cannot be modified.

S4. The value of a constant cannot be modified by an assignment statement because of the restrictions stating that: 1) the target of an assignment must be a variable (RM 5.2/1); 2) an actual in out or out subprogram or entry parameter must be a variable (RM 6.4.1/3); and 3) an

actual in out generic parameter must be a variable (RM 12.3.1/2). Since these are the only contexts that allow modification of an object's value (directly or indirectly by assignment), and since constants cannot be used in these contexts, it follows that a constant's value cannot be modified by an assignment operation.

S5. A task object is a variable even though it cannot be assigned to. This effect of the definitions in RM 3.2.1/2-3 is intentional, to allow task objects (and other objects having a limited type) to be passed as in out parameters of subprograms, entries, and generic units.

S6. Variables can be updated automatically by the underlying hardware when certain events such as I/O transfers occur. Such updates do not contradict the statement in RM 3.2.1/3 that assignment is the only way to update the value of a variable because implicit updates by a system-defined task are allowed if the variable is marked as a shared variable (see RM 9.11/9). An implementation cannot assume that the value of a shared variable is unchanged between assignments made within a particular task.

S7. Although declarations are elaborated in the order of their occurrence (RM 3.9/3 and RM 7.2/3), it is not generally efficient for an implementation to allocate space for an object just when its declaration is elaborated. It is generally more efficient to allocate all needed space in a single operation. This is possible if all expressions occurring (implicitly or explicitly) in an object declaration are free of side effects, as is often the case. But when expressions in object declarations have side effects, the needed space cannot be easily computed in advance because the rules specify the order in which the expressions are to be evaluated. For example, consider:

```

X : INTEGER range 1 .. F+1 := F;           -- F = 2, then 3
Y : constant STRING := (1..F+X => ' ');    -- F = 4
Z : STRING (1..F);                         -- F = 5

```

If F returns successive integer values starting with the value 2, then X will have the subtype INTEGER range 1..3 with initial value 3. Y will have the subtype STRING (1..7), and Z, subtype STRING (1..5). The space for Y cannot be computed until both Fs in X's declaration have been evaluated. In particular, it would be incorrect to evaluate Y's initialization expression (to determine Y's subtype, and consequently, the space needed to hold Y) before evaluating X's initialization expression. (Initialization expressions are evaluated before objects are created (see RM 3.2.1/6) because in the case of unconstrained array and record constants, the initial value determines the subtype of the constant and, therefore, how much space is needed.)

S8. More than one default expression can be specified for a subcomponent:

```

type INNER is
  record
    A, B : INTEGER := 3;
  end record;

type OUTER is
  record
    C : INNER := (4, 5);
  end record;

X : OUTER;

```

RM 3.2.1/14 specifies that default initializations are considered from the outermost composite type to the innermost. Thus, in the above example, X.C.A equals 4, and the default expression given for components A and B is not evaluated.

S9. If an object declaration's subtype indication denotes an unconstrained record type that has

default discriminants, the default expressions are not evaluated if an explicit initialization expression is given (RM 3.2.1/6).

S10. If default expressions are specified for several components of a record, RM 3.2.1/6 says all the expressions must be evaluated before any values are assigned to a component. Even so, constraint checks for a component can be performed as each default expression is evaluated; the requirement to check that an initial value belongs to a component's subtype is stated independently of the rule specifying when assignments are to be performed (see RM 3.2.1/16, which specifies the check, and RM 3.2.1/15 which specifies the sequence of operations). In particular, the required check is not a part of the assignment operation. Consequently, these checks can be performed any time prior to assignment of an initial value. For example:

```

type R is
  record
    A, B, C : INTEGER range 0 .. 1 := F;
  end record;

X : R;

```

Assuming F returns the values 1, 2, and 3 on successive calls, CONSTRAINT_ERROR can be raised after the second call, preventing any third call from being made.

S11. RM 3.2.1/16 requires that initial values of subcomponents belong to the subtype of the component. For components having an array type, this means no subtype conversion is performed:

```

type S is
  record
    STR : STRING (2..3) := "AB";
  end record;

X1 : STRING (2..3) := "AB";    -- no CONSTRAINT_ERROR
X2 : S;                       -- CONSTRAINT_ERROR raised

```

No CONSTRAINT_ERROR is raised for X1 because a subtype conversion is applied to the initial value, "AB", before checking that it belongs to the subtype STRING (2..3). The subtype conversion gives "AB" the bounds 2..3. No such conversion is performed for the initialization of X2.STR, and since "AB" has bounds 1..2 (see RM 4.2/3 and RM 4.3.2/9), CONSTRAINT_ERROR will be raised. This shows that the := used to define default initial values of components does not imply exactly the semantics of the assignment statement. (Note: the same effect holds for := used to define default initial values of subprogram parameters. Moreover, the aggregate (STR => "AB") will raise CONSTRAINT_ERROR since no subtype conversion is applied to "AB" (RM 4.3.2/11).)

S12. RM 3.2.1/16 does not require a subtype conversion for array constants declared with an unconstrained array type since the subtype is determined by the initial value.

S13. A scalar variable can be given an implicit initial value (see RM 3.2.1/17) if the variable is a subcomponent with a specified default value.

S14. RM 3.2.1/18 says "The execution of a program is erroneous if it attempts to evaluate a scalar variable with an undefined value." This means that an implementation can assume that a scalar variable always has a value that satisfies its constraint (because an out-of-range value can never be assigned to such a variable either by an assignment statement, by default initialization, or when returning from a subprogram with a scalar in out or out parameter). The use of a scalar variable as an in out actual generic parameter, or as an out subprogram parameter, is never erroneous, even if the variable has an undefined value when an instantiation of the generic unit is elaborated or when the subprogram is called. Such uses are

not erroneous because only the *names* of such actual parameters are evaluated (RM 6.4.1/4, RM 6.4.1/7, RM 12.3/17, and AI-00365). (For example, if the variable is given as the indexed component A(I), only the object denoted by A(I) is determined; its value is not considered.) The use of an undefined scalar variable as a subprogram in out parameter is, however, erroneous, because the value must be checked against the formal parameter's constraint (RM 6.4.1/5-8).

S15. A constant having an array or record type can have an undefined scalar component:

```
R1 : REC;                      -- undefined component values
C1 : constant REC := R1;
```

An attempt to reference a scalar component of C1 is intended to be considered erroneous (AI-00374).

S16. Evaluation of the name of a scalar variable or constant is not erroneous (even if the variable has an undefined value) if the name occurs as the prefix for the attributes ADDRESS, FIRST_BIT, LAST_BIT, POSITION, or SIZE. In these cases the value of the prefix is not needed (AI-00155):

```
X : INTEGER;
Y : INTEGER := X'SIZE;        -- not erroneous
```

S17. Although RM 3.2.1/18 says the execution of a program is erroneous if it attempts to apply a predefined operator to a variable that has a scalar subcomponent with an undefined value, it is not erroneous to apply a logical operator to an array variable having undefined components if the operands of the logical operator have different lengths; in such a case, CONSTRAINT_ERROR is raised before any array component is evaluated (AI-00426). For example, execution of the following program is not erroneous:

```
type ARR_TYPE is array (INTEGER range <>) of BOOLEAN;
type R (D : INTEGER) is
  record
    A : ARR_TYPE (1 .. D);
  end record;
R1 : R(1);
R2 : R(2);
R3 : R(2) := R1.A or R2.A;    -- CONSTRAINT_ERROR is raised (AI-00426)
```

S18. The assignment of an array object is not erroneous even if some or all subcomponents have undefined scalar values since assignment is not an operator (RM 3.3.3/4). The use of relational operators, logical operators, and catenation is erroneous, however, if either operand has an undefined scalar subcomponent. This means that implementations can assume that values of subcomponents always satisfy the corresponding subcomponent constraints whenever these operators are used.

S19. Although it is convenient to speak of "the use" of a variable or an operator as erroneous, strictly speaking, it is only the evaluation of such a variable or an operator that is erroneous; a program cannot be rejected as erroneous unless an evaluation actually occurs:

```
declare
  X : INTEGER;
begin
  if FALSE then
    Y := X + 1;
  end if;
end;
```

A program containing this block cannot be called erroneous because the statement requiring evaluation of an undefined scalar variable can never be executed.

S20. The evaluation of the attribute X'SIZE is never erroneous, even if X has an undefined value when the attribute is evaluated. The evaluation of the attribute requires an evaluation of the name serving as the prefix, and an evaluation of a simple name just determines the denoted object (RM 4.1/9).

S21. If the full declaration of a private type declares a scalar type and a variable of the private type has an undefined value, an equality comparison using the variable will be erroneous because the predefined equality for the private type is the predefined equality for the type actually used to represent the private type. Since the actual type is a scalar type, RM 3.2.1/18 will apply.

S22. If a record type with default discriminants is used as the subtype indication in an object declaration for a variable, each default discriminant value must be checked for compatibility with its use within the declared object, as specified in RM 3.7.2/5 (see AI-00308). For example:

```
type R (D : INTEGER := -1) is
  record
    COMP : STRING (D .. 10);
  end record;

X : R;                                -- CONSTRAINT_ERROR
```

When the default value of the discriminant is checked for compatibility with its use in declaring X.COMP, CONSTRAINT_ERROR will be raised since the index range -1..10 is not null and -1 does not belong to STRING's index subtype, POSITIVE (RM 4.3.2/11). Similarly, consider:

```
type R2 (D : INTEGER := -2) is
  record
    case D is
      when -10 .. -2 =>
        C1 : R(D);
      when others =>
        C2 : INTEGER;
    end case;
  end record;

Y : R2;                                -- CONSTRAINT_ERROR
```

CONSTRAINT_ERROR is raised because the default discriminant value, -2, is used. Y contains component C1 when the discriminant value is -2. Since -2 is not a compatible discriminant value for this component, CONSTRAINT_ERROR is raised. If the default discriminant value had been -1, then the initial value of Y would not have contained component C1 and no exception would be raised (see IG 3.7.3/S and AI-00358).

Changes from July 1982

S23. Dependence on the order of evaluation of subcomponent initializations is no longer erroneous. (This change means that implementations cannot assume that the same value is produced for each initialization expression, no matter in what order the expressions are evaluated.)

S24. Only scalar variables and scalar components can have undefined values. All composite objects are considered as having defined values.

Changes from July 1980

S25. Only an attempt to evaluate a scalar variable (including a scalar subcomponent) is considered erroneous.

Approved Interpretations

S26. When the discriminant of an object is determined by default, `CONSTRAINT_ERROR` is raised if a discriminant value is not compatible with the type of the object (AI-00308). See IG 3.7.2/S for further discussion of such compatibility checks.

S27. A constant can have an undefined scalar component. An attempt to use the value of such a component is considered erroneous (AI-00374).

S28. Evaluation of the name of a scalar variable (or constant) having an undefined value is not erroneous if the name occurs as the prefix for the attribute `ADDRESS`, `FIRST_BIT`, `LAST_BIT`, `POSITION`, or `SIZE` (AI-00155).

S29. If the operands of a predefined logical operator do not have the same number of components, the execution of a program is not erroneous; `CONSTRAINT_ERROR` is raised (AI-00426).

Legality Rules

- L1. The base type of an initialization expression in an object declaration must be the same as the base type of the object being initialized (RM 3.2.1/1).
- L2. Object declarations containing the word **constant** must have an initialization expression (RM 3.2.1/2), except for deferred constant declarations (i.e., except when the object being declared has a private or limited private type, the declaration appears in the package that declares the private type, and the declaration appears prior to the full declaration of the private type (RM 7.4/4)).
- L3. An initialization expression is not allowed for objects having a limited type (RM 7.4.4/6).
- L4. A constant declaration is not allowed in a package if: 1) its subtype indication is a private type with a discriminant constraint; 2) the declaration occurs in the package that declares the private type, and the constant declaration occurs before the full declaration of the private type (RM 7.4/4). (Such a declaration is an illegal deferred constant declaration.)
- L5. An identifier declared in an object declaration must not have been declared in any preceding declaration of the same declarative region, except that the full declaration of a deferred constant must be preceded by a deferred constant declaration for the same identifier (RM 8.3/5 and RM 7.4.3/1).
- L6. No two identifiers in an object declaration's identifier list can be identical (RM 3.2/10 and RM 8.3/5).
- L7. The use of a name denoting a declared object is not allowed in the object's subtype indication, constrained array definition, or initialization expression (RM 8.3/5).

Exception Conditions

- E1. For an object declaration that specifies an initial value and that declares:
- a scalar variable or constant, `CONSTRAINT_ERROR` is raised if the initial value lies outside the range specified by the subtype indication (RM 3.2.1/16, RM 3.3/4, and RM 3.5/3).
 - a non-null constrained array variable or constant, `CONSTRAINT_ERROR` is

raised if corresponding dimensions of the initial value and the object do not have the same length (RM 3.2.1/16 and RM 5.2/2).

- a null constrained array variable or constant, `CONSTRAINT_ERROR` is raised if the initial value is not a null array value.
 - a constrained variable or constant with discriminants, `CONSTRAINT_ERROR` is raised if corresponding discriminants of the initial value and the object do not have the same value (RM 3.2.1/16, RM 3.3/4, and RM 3.7.2/6).
 - a constrained access variable or constant, `CONSTRAINT_ERROR` is raised if the initial value is not null, the designated type:
 - is an array type, and the index bounds of the object designated by the initial value do not equal those imposed by the access subtype (RM 3.2.1/16, RM 3.3/4, RM 3.8/6, and RM 3.6.1/4).
 - has discriminants and the discriminant values of the object designated by the initial value do not equal those imposed by the access subtype (RM 3.2.1/16, RM 3.3/4, RM 3.8/6, and RM 3.7.2/6).
- E2. For an object declaration that does not specify an initial value, `CONSTRAINT_ERROR` is raised if any default value (for a subcomponent) does not belong to the subtype of the subcomponent being initialized (see IG 3.7/E for further details) (RM 3.2.1/16).
- E3. For an uninitialized object declaration whose subtype indication denotes an unconstrained type with discriminants that have defaults, `CONSTRAINT_ERROR` is raised if any default discriminant value does not belong to the discriminant's subtype or if the value is not compatible with its use within the record. (The value is only checked for subcomponents that exist for the subtype defined by the default value, and only for subcomponents whose subtype definition depends on a discriminant (AI-00358)) (AI-00308; see also IG 3.7.2/S).
- E4. `CONSTRAINT_ERROR` can be raised when the subtype indication or constrained array definition is elaborated. See IG 3.3.2/E and IG 3.6.1.b/E for further information.
- E5. `STORAGE_ERROR` is raised by an object declaration if there is insufficient storage available to hold the object (RM 11.1/8).

Test Objectives and Design Guidelines

- T1. Check that objects having a limited private type or a subcomponent of a limited private type cannot be given initial values.
- Check that objects having a task subcomponent cannot be given initial values (see IG 9.2/T1).
- T2. Check that a task object cannot be given an initial value (see IG 9.2/T1).
- T3. Check that constant object declarations must have an explicit initialization expression.
- Implementation Guideline:* Check for all nonlimited types.
- Implementation Guideline:* Include a case when a default value exists for every component of the object.
- T4. Check that unconstrained array definitions are not permitted in object declarations.
- T5. Check that an identifier declared by an object declaration cannot have been declared previously in the same declarative region (see IG 8.3/T1-T8).
- Check that an identifier denoting a declared object cannot be used in its own declaration (see IG 8.3/T11).
- T6. Check that if several identifiers are declared in the same object declaration with an array type definition, they each have a unique type.

Check that objects declared with an array type definition in separate object declarations each have a unique type.

- T7. Check that object declarations are elaborated in the order of their occurrence, i.e., that expressions associated with one declaration (including default expressions, if appropriate) are evaluated before any expression belonging to the next declaration.

Implementation Guideline: Include a check for objects having a generic formal type with default values.

Check that expressions in the subtype indication or the constrained array definition are evaluated before any initialization expressions are evaluated.

Implementation Guideline: Include a case where no explicit initial value is provided and the subtype indication contains an index or discriminant constraint.

Implementation Guideline: Include a case where the constrained array definition's component declaration contains an index or discriminant constraint.

- T8. Check that if an explicit initialization expression is given for an object declaration, no default expressions are evaluated.

Implementation Guideline: Include a check for default discriminant expressions.

Check that if a default expression is evaluated for a component, no default expressions for any subcomponents are evaluated.

Implementation Guideline: Include a check for a generic formal type.

Check that if a discriminant constraint is given and if no initial value is specified, default expressions for the discriminants are not evaluated, but default expressions for other components are evaluated.

Implementation Guideline: Include a check for a generic formal type.

- T11. Check that when a variable or constant having an enumeration, integer, float, or fixed type is declared with an initial value, `CONSTRAINT_ERROR` is raised if the initial value lies outside the range of the subtype.

Implementation Guideline: Check for a generic formal type (see IG 12.3.3/T5).

- T12. Check that when a variable or constant having a non-null array subtype is declared with an initial value, `CONSTRAINT_ERROR` is raised if corresponding dimensions of the initial value and the subtype do not have the same length.

Implementation Guideline: Check for generic formal types also.

Check that `CONSTRAINT_ERROR` is raised for the declaration of a null array object if the initial value is not a null array.

- T13. Check that when a variable or constant having a constrained type with discriminants is declared with an initial value, `CONSTRAINT_ERROR` is raised if corresponding discriminants of the initial value and the subtype do not have the same value.

Implementation Guideline: Use both record types and private types with discriminants.

Implementation Guideline: Include a check for a generic formal type with discriminants.

- T14. Check that if the subtype indication in an object declaration denotes an unconstrained type with default discriminants, and no explicit initial value is provided in the object declaration, `CONSTRAINT_ERROR` is raised if a default discriminant value is not compatible with the discriminant's subtype or with its use within the declared object (see IG 3.7.2/T13-T16).

- T15. Check that when a variable or a constant having a constrained access type is declared with an initial non-null access value, `CONSTRAINT_ERROR` is raised if an index bound or a discriminant value of the designated object does not equal the corresponding value specified for the access subtype.

- T16. Check that when a variable having a record or private type is declared without an explicit

initialization, `CONSTRAINT_ERROR` is raised if a default value for a subcomponent does not belong to the component's subtype (see IG 3.7/T8).

- T17. Check whether all default initialization expressions are evaluated before any value is checked to see if it belongs to a component's subtype.

Implementation Guideline: Try one case where an out-of-range default discriminant value is used later to define a component subtype, and a case where a non-discriminant default value does not belong to its component subtype.

Implementation Guideline: Repeat the check for a generic formal type.

3.2.2 Number Declarations

Semantic Ramifications

S1. In many ways, the name introduced by a number declaration acts as a macro for a literal having the value of the initialization expression. However, since a number declaration is stated to be "a special form of object declaration" (RM 3.2/8), a named number is considered to be an object. This means that an address clause can be specified for a named number:

```
ONE : constant := 1;
for ONE use at ...;
```

if an implementation accepts this representation clause, then it means that storage is provided to hold the specified value at the specified address. Of course, such an address clause would be illegal for a literal, since a simple name denoting an object must be used (RM 13.5/3-4), and a literal is not a simple name.

S2. A named number can be used where its equivalent literal could not be used:

```
M1 : constant := -1;
...
for I in M1 .. 10 loop      -- legal
for I in -1 .. 10 loop     -- illegal
```

The signed literal is illegal when the other bound of the range in a loop is also a literal (see IG 3.6.1.a/S and RM 3.6.1/2), but the named number can be safely used in place of the literal.

S3. The type of a named number is determined by the type of the initialization expression, and this type is determined by the form of the literals and attributes used in the expression. See IG 4.10/S for further discussion.

Changes from July 1982

S4. There are no significant changes.

Changes from July 1980

S5. There are no significant changes.

Legality Rules

- L1. The initializing expression in a number declaration must be a static expression having the type *universal_integer* or *universal_real* (RM 3.2.2/1).
- L2. An identifier declared in a number declaration must not have been declared in any preceding declaration of the same declarative region (RM 8.3/15 and RM 7.4.3/1).
- L3. No two identifiers in a number declaration's identifier list can be identical (RM 3.2/10 and RM 8.3/15).

- L4. Use of a name denoting a named number is not allowed in the initialization expression for the named number (RM 8.3/5).

Test Objectives and Design Guidelines

- T1. Check that the following attributes cannot appear in a number declaration because:

- they do not return *universal_integer* or *universal_real* values: ADDRESS, BASE, CONSTRAINED, FIRST (for a scalar or array subtype), IMAGE, LAST (for a scalar or array subtype), MACHINE_OVERFLOWS, MACHINE_ROUNDS, PRED, PRIORITY, RANGE, SUCC, TERMINATED, and VAL.
- they do not return static values: COUNT, LAST_BIT, LENGTH, FIRST_BIT, POSITION, and STORAGE_SIZE.
- they do not return static values when their prefixes are not static types: POS and SIZE.
- the returned value is not static if the argument is not static: POS.

Check that a user-defined function, operator, or the operator "&" cannot appear in a number declaration.

Check that a string literal or character literal cannot appear in a number declaration.

- T2. Check that an integral number name cannot be used in a context requiring a real value, and a real number name cannot be used in a context requiring an integer value.

Implementation Guideline: Check discrete ranges, initialization expressions, discriminant values, choices in case statements and variant parts, assignment statements, equality comparisons, qualified expressions, actual parameters, and the value of delta or the bound of a range of a real type definition.

- T3. Check that the following attributes can appear in number declarations:

- for prefixes denoting a static scalar type: SIZE.
- for prefixes denoting a static scalar type and for static arguments: POS.
- for static floating point prefixes: DIGITS, EMAX, EPSILON, LARGE, MACHINE_EMAX, MACHINE_EMIN, MACHINE_MANTISSA, MACHINE_RADIX, MANTISSA, SAFE_EMAX, SAFE_LARGE, SAFE_SMALL, SMALL.
- for static fixed point prefixes: AFT, DELTA, FORE, LARGE, MANTISSA, SAFE_EMAX, SAFE_LARGE, SAFE_SMALL, SMALL.

Check that certain SYSTEM constants can appear in number declarations: FINE_DELTA, MAX_DIGITS, MAX_INT, MAX_MANTISSA, MEMORY_SIZE, MIN_INT, TICK, and STORAGE_UNIT.

- T4. Check that an identifier in a number declaration cannot have been declared previously in the same declarative region (see IG 8.3/T1-T8).

Check that the declared identifier cannot be used in the initialization of its own declaration (see IG 8.3/T11).

- T5. Check that integer and real literals, as well as named numbers, can be used in the declaration of named numbers (see IG 4.10/T).

3.3 Types and Subtypes

Semantic Ramifications

S1. Scalar types are not the only types whose values have no components. Values of access types and private types without discriminants also have no components.

S2. The classes of types defined in RM 3.3/2 are mutually exclusive and collectively exhaustive, i.e., every type that can be declared in Ada belongs to exactly one of these classes. However, other classes also exist:

- *numeric* types are integer types and real types (RM 3.5/1).
- *real* types are fixed point types and floating point types (RM 3.5.6/1).
- *discrete* types are integer and enumeration types (RM 3.5/1).
- *limited* types are limited private types, types having a component of a limited type, and task types (RM 7.4.4/2).

S3. The name declared by a type declaration sometimes denotes a constrained base type (a subtype) rather than a base type (see IG 3.3.1/S).

S4. Since a type is a subtype of itself (RM 3.3/4), the terms type and subtype are, strictly speaking, interchangeable. The RM, however, tends to use the term "type" to mean base type, and the term "subtype" to mean a constrained base type or a type mark declared with a subtype declaration.

S5. The visibility rules forbid certain forms of recursive type declaration, namely, those in which a type mark is used in its own declaration (RM 8.3/5):

```

type T is
  record
    A : T;           -- illegal; T not yet visible
  end record;

type U is array (1..10) of U; -- illegal; U not yet visible

type R is digits R'DIGITS;   -- illegal; R not yet visible

task type S is
  entry START (X : S);       -- illegal; S not yet visible
end S;
```

S6. Private types and incomplete types present additional possibilities for illegal recursive type definitions:

```

package Q is
  type P0 is private;

  type P1 is private;
  subtype SP1 is P1;

  type P2 is private;
  subtype SP2 is P2;
private
```

```

type P0 is access P0;    -- illegal; 8.3/5
type P1 is new SP1;      -- illegal; 7.4.1/4
type P2 is access SP     -- legal

type V;
type V is access V;      -- illegal; 8.3/5
end Q;

```

The full declaration of P0 is illegal because the previous declaration of P0 is hidden and the current declaration is not yet visible (RM 8.3/5). The same reasoning makes the full declaration of incomplete type V illegal. The full declaration of P1 is illegal because RM 7.4.1/4 forbids the use of any name denoting an incompletely declared private type in a derived type definition. The full declaration of P2 is allowed because the declaration of SP2 is visible.

S7. Indirect forms of recursive type declarations are forbidden by RM 3.3/8. For example, the full declaration of R2, below, is illegal because of this rule:

```

package P1 is
  type R1 is private;
  type R2 is private;
private
  type R1 is
    record
      ER1 : R2;
    end record;
  type R2 is
    record
      ER2 : R1;    -- illegal
    end record;
end P1;

```

Similar examples can be constructed using array type declarations as the full declarations of R1 and R2. Also, it would be illegal if R2's full declaration were a derived type declaration:

```

type R2 is new R1;    -- illegal

```

S8. Now consider more complex examples:

```

package P is
  type T is private;          -- (1)
  subtype ST is T;
private
  type A is array (1..0) of ST; -- (2); builds on (1)
  type R (D : BOOLEAN := TRUE) is
    record
      case D is
        when FALSE =>
          null;
        when TRUE =>
          C : A;    -- (3); builds on (2)
        end case;
    end record;

```

```

-- Now consider alternate full declarations of T that have components
-- of type T, and so are illegal.
type T is new A;                                -- illegal
type T is array (1..0) of ST;                    -- illegal

type T is array (1..0) of R;                      -- illegal
type T is array (1..0) of R(FALSE);              -- illegal
type T is new R(FALSE);                          -- illegal

```

The constrained array type declarations are illegal even though the only allowed value is a null array (i.e., even though the only allowed value has no component), since T is a subtype, not a base type (RM 3.6/6-8). Similarly, the last two declarations are illegal even though R(FALSE) has no subcomponents of type T (R's base type has a subcomponent of type T).

S9. The following declarations are legal, although not very useful:

```

type M;
type AM is access M;

type M is
  record
    E : AM;                                -- legal
  end record;

type W;
type X is access W;
type W is access X;

task type T;
subtype U is T;
task body T is
  X : U;                                -- legal

```

The declaration of these types is legal because the types M, X, W, and T do not have any subcomponents of type M, X, W, and T, respectively. (Of course, after a task of type T is activated, another task of type T will have been created, and its activation will in turn create a third task, etc. Therefore, although legal, the activation of a task of type T will cause an unlimited number of task objects to be created.)

S10. For a scalar subtype, a value belongs to the subtype if it lies in the range specified for the subtype (since this is what it means to satisfy a range constraint; RM 3.5/3). An array value belongs to a constrained array subtype if it has the bounds specified for the subtype's index constraint (RM 3.6.1/4). A value of a type with discriminants belongs to a constrained subtype if corresponding discriminant values of the value and of the subtype are equal (RM 3.7.2/6). An access value belongs to an access subtype if the value is null or if the designated object satisfies the constraint imposed on the designated type (RM 3.8/6).

Changes from July 1982

S11. There are no significant changes.

Changes from July 1980

S12. The term subcomponent is introduced and defined.

Legality Rules

- L1. No type can be declared so that a subcomponent of the base type would have its own type (RM 3.3/8).

Test Objectives and Design Guidelines

- T1. Check that the type declared by a type declaration cannot have subcomponents of the declared type.

Implementation Guideline: Check recursive type declarations using record, array, and private types.

- T2. Check that a type mark cannot be used in its own declaration.
- T3. Check that certain forms of almost recursive types can be declared — in particular, a record having a component of an access type whose designated type is the record type; an access type whose designated type is an access type that designates the first access type; and similarly, a private type whose full declaration declares an access type designating the private type.

3.3.1 Type Declarations**Semantic Ramifications**

S1. The name declared by a full type declaration is not necessarily the name of a base type. The name declared for a numeric or derived type denotes a subtype of an anonymous base type (RM 3.4/1, RM 3.5.4/5, RM 3.5.7/11, and RM 3.5.9/9). The name declared by a constrained array type declaration declares both an array base type and an array subtype (RM 3.6/6). Both the subtype and the base type are declared by such full type declarations.

S2. `NUMERIC_ERROR` (or `CONSTRAINT_ERROR`; see AI-00387) can be raised when an array type definition is elaborated for a constrained array type if the number of components in a dimension exceeds the largest value of the index base type. See IG 3.6.1/S for further discussion.

Approved Interpretations

- S3. `CONSTRAINT_ERROR` can be raised instead of `NUMERIC_ERROR` (AI-00387).

Changes from July 1982

- S4. There are no significant changes.

Changes from July 1980

- S5. There are no significant changes.

Legality Rules

- L1. If a discriminant part is given in a full type declaration, the type definition must be a record type definition (RM 3.7.1/3).
- L2. The identifier declared in an incomplete type declaration or a private type declaration must not have been declared in any preceding declaration of the same declarative region (RM 8.3/15).
- L3. The identifier declared in a full type declaration must not have been declared in any preceding declaration of the same declarative region except that the full declaration of an incomplete or private type must be preceded by an incomplete or private type declaration for the same identifier (RM 8.3/15, RM 7.4.1/1, and RM 3.8.1/3).

Exception Conditions

- E1. **NUMERIC_ERROR** (or **CONSTRAINT_ERROR**; see AI-00387) can be raised when a constrained array type declaration is elaborated if the number of components specified for a dimension lies outside the range of the index base type (see RM 11.1/6, IG 3.6.1/S and AI-00387).

Test Objectives and Design Guidelines

- T1. Check that a discriminant part is not allowed in a type declaration if the rest of the declaration is an enumeration, integer, real, array, access, or derived type definition.
- T2. Check that the identifier declared by a type declaration, incomplete type declaration, private type declaration, or subtype declaration cannot have been declared previously in the same declarative region.

Implementation Guideline: Check for all forms of declarative regions: within a single package specification, within a package specification and its body, within a package body, within the declarative part of a block, within the declarative part of a subprogram body (include type declarations that duplicate a formal parameter name), within a generic unit declaration (for type declarations that duplicate identifiers declared as formal parameters of the generic unit and of a generic subprogram).

Check that two full declarations cannot be given for the same incomplete or private type.

Implementation Guideline: For the incomplete type case, check that if one full declaration is given in the private part of a package specification, another cannot be given in the package body.

Implementation Guideline: Check for both generic and non-generic units.

3.3.2 Subtype Declarations

Semantic Ramifications

- S1. The rules of the language are such that non-scalar constrained subtypes cannot be further constrained, but subtypes of scalar types can be further constrained, e.g.:

```
subtype SMALL is POSITIVE range 1..10;
subtype SMALLER is SMALL range 2..5;
```

The compatibility requirement for scalar types (see RM 3.5/4) ensures that the bounds of a range constraint cannot "widen" the range of a subtype:

```
subtype BIGGER is SMALL range 2..11;      -- incompatible constraint
```

Because the constraint is incompatible with the range specified for subtype **SMALL**, **CONSTRAINT_ERROR** will be raised (RM 3.3.2/9).

Changes from July 1982

- S2. There are no significant changes.

Changes from July 1980

- S3. There are no significant changes.

Legality Rules

- L1. If a range constraint appears after a type mark in a subtype indication, the type mark must denote a scalar base type or subtype (RM 3.5/4), and the subtype indication must not appear in an allocator (RM 4.8/4).
- L2. If a floating point constraint appears after a type mark in a subtype indication, the type mark must denote a floating point base type or subtype (RM 3.5.7/14), and the subtype indication must not appear in an allocator (RM 4.8/4).

- L3. If a fixed point constraint appears after a type mark in a subtype indication, the type mark must denote a fixed point base type or subtype (RM 3.5.9/13), and the subtype indication must not appear in an allocator (RM 4.3.1).
- L4. If an index constraint appears after a type mark in a subtype indication, the type mark must denote an unconstrained array type or an access type whose designated type is an unconstrained array type; the index constraint must provide a discrete range for each index of the array type, and the base type of each discrete range must be the same as that of the corresponding index (RM 3.6.1/3).
- L5. If a discriminant constraint appears after a type mark in a subtype indication, the type mark must denote an unconstrained (record or private) type with discriminants or an access type whose designated type is an unconstrained (record or private) type with discriminants; exactly one value must be specified for each discriminant, and each value must have the type of the corresponding discriminant (RM 3.7.2/1 and RM 3.7.2/4).
- L6. The identifier declared by a subtype declaration must not have been declared in any preceding declaration of the same declarative region (RM 8.3/15).

Exception Conditions

- E1. **CONSTRAINT_ERROR** is raised if the constraint specified in a subtype indication is not compatible with the condition specified by the type mark (RM 3.3.2/9). (Detailed compatibility checks are specified in the sections for each class of type for which a constraint is allowed.)

Test Objectives and Design Guidelines

Implementation Guideline: For objectives T1-T5, check all contexts in which a subtype indication can appear: access type definition, allocator, component of a record or array type declaration, derived type declaration, and object declaration. Include the use of constrained array type definitions and access type definitions in generic formal parameter declarations.

- T1. Check that in a subtype indication, a range constraint is not allowed if the type mark denotes an array, record, access, task, or private type.
- T2. Check that in a subtype indication, a floating point constraint is not allowed if the type mark denotes an enumeration, integer, fixed point, array, record, access, task, or private type.
- T3. Check that in a subtype indication, a fixed point constraint is not allowed if the type mark denotes an enumeration, integer, floating point, array, record, access, task, or private type.
- T4. Check that in a subtype indication, an index constraint is not allowed if the type mark denotes an enumeration, integer, floating point, fixed point, constrained array, record, constrained access, task, or private type; also, an access type whose designated type is one of these types.

Implementation Guideline: IG 4.8/T2 checks the use of such subtype indications in an allocator.

- T5. Check that in a subtype indication, a discriminant constraint is not allowed if the type mark denotes an enumeration, integer, floating point, fixed point, array, constrained record, constrained access, task, or private type; also, an access type whose designated type is one of these types.

Implementation Guideline: IG 4.8/T2 checks the use of such subtype indications in an allocator.

- T6. Check that the identifier declared by a subtype declaration cannot have been declared earlier in the same declarative region (see IG 3.3.1/T2).
- T21. Check that a subtype can be declared by a subtype declaration (checked implicitly by other tests).

3.3.3 Classification of Operations

Semantic Ramifications

S1. T'BASE'BASE is a legal prefix of an attribute since the prefix of the second BASE is T'BASE, which denotes a type.

S2. T'BASE'BASE'FIRST is legal if T'BASE'FIRST is legal since the base type of a base type is the same base type (RM 3.3/4); T'BASE'BASE names the base type of T'BASE, which is already a base type (the base type of T). Since T'BASE'BASE and T'BASE name the same type, if T'BASE'FIRST is legal, so is T'BASE'BASE'FIRST, and both have the same meaning.

S3. The declaration:

```
C : array (1..3) of INTEGER := 1 & 2 & 3;
```

is legal because it is equivalent to the following sequence of declarations (RM 3.6/6-8):

```
subtype anon1_5 is INTEGER range 1..5;
type anon_arr is array (anon1_5 range <>) of INTEGER;
-- implicit declaration of array operations, including & operations
C : anon_arr (1..3) := 1 & 2 & 3;
```

However, the following is illegal:

```
package P is
  type T is range 1 .. P."+"(1, 1); -- illegal
```

RM 3.5.4/4-5 says the type declaration is equivalent to:

```
package P is
  type anon is new predefined_integer_type;
  subtype T is anon range 1 .. P."+"(1, 1);
```

The implicit declarations of integer operations do not, however, occur immediately after the implicit type declaration because RM 3.3.3/2 says these operations are declared "immediately after the type definition", i.e.,

```
package P is
  type T is range 1 .. P."+"(1, 1); -- illegal
  -- implicit declarations of integer operations
```

Since the operations are declared after the type definition, there is no visible "+" operation declared within P that can be referenced by P."+", and so the name P."+ is illegal.

S4. The following examples are illegal for the same reason: the referenced operation is declared after the full type declaration:

```
package Q is
  type T is digits 3;
  function F return T;
end Q;

with Q;
package P1 is
  type T1 is new FLOAT range 1.0 .. P1."+"(1.0, 1.0); -- illegal
  -- premature use of "+" declared for T1
```



```

type T2 is new Q.T range 0.0 .. Q.T(P1.F);           -- illegal
-- premature use of derived subprogram F

type T3 is new BOOLEAN range FALSE..BOOLEAN (P1.TRUE); -- illegal
-- premature use of derived enumeration literal

type T4 is private;           -- declares "=" for T4
type T5 is private;           -- declares "=" for T5
type T6 is private;

private
type T4 is range 1 .. BOOLEAN'POS (P1."="(1,1));      -- illegal
-- premature use of implicit conversion to type T4

type T5 is access BOOLEAN range
      FALSE .. P1."="(new BOOLEAN, null); -- illegal
-- premature use of allocator and null literal operations

type T6 is array (FALSE .. P1."="( "AB", ('A', 'B'))) -- illegal
      of CHARACTER;
-- premature use of string literal and aggregate operations
end P1;

```

S5. Component selection operations are declared after a record type definition.

```

package P2 is
  type T7 (D : INTEGER) is private;
  -- selection operation for component D implicitly declared here
private
  function F return T7;
  type T7 (D : INTEGER) is
    record
      C1 : INTEGER := F.D;           -- legal
      C2 : INTEGER := F.C1;         -- illegal
    end record;
  -- selectors for .C1 and .C2 implicitly declared here
end P2;

```

The use of F.D in the initialization of component C1 is legal because the selection operation for component D has been declared and is visible, and the use of the name of discriminant D is explicitly allowed within the record type definition for T7 (RM 3.7/3). The use of F.C1 is illegal, however, for two reasons: the component selection operation for C1 has not yet been declared, and RM 3.7/3 rules out the use of a component name (other than a discriminant component) within its containing record type definition.

Changes from July 1982

S6. The place where a type's operations are implicitly declared is explicitly defined.

Changes from July 1980

S7. The attribute 'BASE is defined.

S8. The class of basic operations is defined.

Legality Rules

L1. An attribute having the form prefix'BASE must be used as the prefix in the name of another attribute, and the prefix of the BASE attribute designator must denote a type or subtype.

- L2. If T denotes an enumeration type, the name T'BASE can only be used as a prefix of one of the following attributes: BASE, FIRST, IMAGE, LAST, POS, PRED, SIZE, SUCC, VAL, VALUE, WIDTH.
- L3. If T denotes an integer type, the name T'BASE can only be used as a prefix of one of the following attributes: BASE, FIRST, IMAGE, LAST, POS, PRED, SIZE, SUCC, VAL, VALUE, WIDTH.
- L4. If T denotes a floating point type, the name T'BASE can only be used as a prefix of one of the following attributes: BASE, DIGITS, EMAX, EPSILON, FIRST, LARGE, LAST, MACHINE_EMAX, MACHINE_EMIN, MACHINE_MANTISSA, MACHINE_OVERFLOW, MACHINE_RADIX, MACHINE_ROUNDS, MANTISSA, SAFE_EMAX, SAFE_LARGE, SAFE_SMALL, SIZE, SMALL.
- L5. If T denotes a fixed point type, the name T'BASE can only be used as a prefix of one of the following attributes: AFT, BASE, DELTA, FIRST, FORE, LARGE, LAST, MACHINE_OVERFLOW, MACHINE_ROUNDS, MANTISSA, SAFE_LARGE, SAFE_SMALL, SIZE, SMALL.
- L6. If T denotes an array type, the name T'BASE can only be used as a prefix of one of the following attributes: BASE, SIZE.
- L7. If T denotes a record type, the name T'BASE can only be used as a prefix of one of the following attributes: BASE, SIZE.
- L8. If T denotes a private type or subtype with discriminants, the name T'BASE can only be used as a prefix of one of the following attributes: BASE, CONSTRAINED, SIZE.
- L9. If T denotes an access type, the name T'BASE can only be used as a prefix of one of the following attributes: BASE, SIZE, STORAGE_SIZE.
- L10. If T denotes a task type, the name T'BASE can only be used as a prefix of one of the following attributes: BASE, SIZE, STORAGE_SIZE.

Test Objectives and Design Guidelines

- T1. Check that T'BASE cannot be used by itself as an attribute in any context requiring a type mark or a name that denotes a type.

Implementation Guideline: The required contexts are:

- for a name that denotes a type: generic actual parameter (when the formal parameter is a private type or an array type).
- for a type mark: in an access type definition, allocator, declaration of a record or array component, derived type definition, discrete range, membership test, object declaration, discriminant specification, generic parameter declaration, specification of an index subtype, formal parameter declaration, qualified expression, renaming declaration, and type conversion.

- T2. Check that operations are declared implicitly after the type definition in a type declaration.

Implementation Guideline: See the examples of illegal type declarations given in the Semantic Ramifications section.

- T11. Check that T'BASE is allowed as the prefix only of attributes suitable for use with the type denoted by T (see IG sections for each class of type and their attributes).

Check that the prefix of 'BASE must denote a type (see IG sections for each class of type and their attributes).

Check that when ST is a subtype, ST'BASE attribute returns the correct values for the base type rather than for the subtype (see IG sections for each class of type and their attributes).

3.4 Derived Type Definitions

Semantic Ramifications

S1. The term "derived type" is defined explicitly in RM 3.4/1 to be a base type. For example:

```
type MY_NATURAL is new NATURAL;
```

NATURAL is the parent subtype; NATURAL's base type, INTEGER, is the parent (base) type. The derived type declared by this type declaration is an anonymous type derived from the parent type, INTEGER. The derived subtype is MY_NATURAL. MY_NATURAL'BASE'FIRST has the same numeric value as NATURAL'BASE'FIRST, but the types of these values are different, since the base types are different. Similarly, MY_NATURAL'FIRST and NATURAL'FIRST have the same value but different types, i.e., INTEGER(MY_NATURAL'FIRST) = NATURAL'FIRST.

S2. Although a derived type declaration declares both a base type and a subtype, a derived type declaration such as:

```
type NT is new INTEGER_TYPE range 1..5;
```

cannot be considered precisely equivalent to:

```
type %NT_BASE% is new INTEGER_TYPE'BASE;
subtype NT is %NT_BASE% range 1..5;
```

The equivalence is not exact because operations implicitly declared for NT's base type are placed after the type declaration that actually occurs in the text, not after some fictional declaration of %NT_BASE% (see RM 3.3.3/2). This makes a difference:

```
package P is
  type T is new BOOLEAN;
  function F return T;
private
  package Q is
    type U is new T range FALSE .. T(Q.F); -- illegal
    -- Q.F is implicitly declared here
  end Q;
end P;
```

Since Q.F is declared implicitly after the type declaration that occurs in the text, it is not visible prior to that point, and so cannot be referenced in the declaration of U.

S3. Similarly, CONSTRAINT_ERROR is raised when the following declaration is elaborated:

```
type T is new POSITIVE range 0 .. 100;
```

CONSTRAINT_ERROR is raised when the subtype indication POSITIVE range 0..100 is elaborated, since zero does not belong to the subtype POSITIVE. CONSTRAINT_ERROR is raised even though the base type of T includes the value zero and even though POSITIVE mainly serves to indicate the parent base type. No exception would be raised if a derived type declaration were considered exactly equivalent to:

```
type %ANON% is new POSITIVE'BASE;
subtype T is %ANON% range 0 .. 100;
```

S4. The RM says a derived type belongs to the same class of types as its parent type (RM 3.4/4). The classes of types mentioned in the RM are: scalar, discrete, enumeration, integer, real, floating, fixed, numeric, composite, array, record, access, private, limited private, limited,

and task. Various rules in the language are defined as applying to these classes of types. Each of these rules, therefore, applies to a derived type whose parent belongs to one of these classes.

S5. All the values of the parent (base) type are included in the set of values of the derived type, e.g., consider:

```
type ENUM is (E1, E2);
type NEW_ENUM is new ENUM range E2 .. E2;
```

The expression

```
NEW_ENUM'BASE'FIRST = E1
```

is legal and raises no exception since E1 is implicitly declared as a value of type NEW_ENUM, and an equality operation is implicitly declared for the base type, ENUM'BASE. It is always possible to create a value of the derived (base) type, e.g.,

```
package P is
  type T (D : INTEGER) is
    record
      C : INTEGER;
    end record;
  subtype T2 is T(2);

  function CREATE (DISC : INTEGER) return T;
end P;

package body P is
  function CREATE (DISC : INTEGER) return T is
  begin
    return (DISC, 3);
  end CREATE;
end P;

with P;
procedure Q is
  type NT is new P.T2;
begin
  if CREATE(2) /= NT'(2,3) then      -- no exception
    ...
  if (3, 3) = NT'(2,3) then         -- no exception
    ...
  end Q;
```

CREATE returns a value belonging to NT's base type (RM 3.4/14); no exception is raised even though the value does not belong to the derived subtype, NT. Similarly, the aggregate, (3, 3), is a value of NT's base type, even though it is not a value that belongs to subtype NT.

S6. The rule for conversion between derived types allows one type to be converted to another if both types are derived from a common ancestor or if one type is derived, directly or indirectly, from the other (RM 4.6/9).

S7. Although predefined operators and enumeration literals are subprograms, they are not considered derived subprograms. For example, when BOOLEAN is declared in STANDARD, the relational operators are implicitly declared to take arguments of type BOOLEAN and return a result of type BOOLEAN:

```

package STANDARD is
...
type BOOLEAN is (FALSE, TRUE);
-- implicit declaration of
-- function "<" (L, R : BOOLEAN) return BOOLEAN;
...
end STANDARD;

```

The relational operators are not derivable functions because they are declared implicitly (RM 3.4/11). When declaring a type derived from BOOLEAN, the relational (and other operators that are predefined for boolean types; see RM 3.5.5/15) are implicitly declared rather than explicitly. For example:

```

package P is
  type NB is new BOOLEAN;
  function "<" (L, R : NB) return NB;
end P;

package Q is
  type NNB is P.NB;
end Q;

```

Among the predefined operators implicitly declared for type NB are (RM 4.5.1/2 and RM 4.5.2/3):

```

function "and" (L, R : NB) return NB;
function "<" (L, R : NB) return BOOLEAN;

```

The types returned by these predefined operators are specified in RM 4.5 for each operator. If these operators had been declared by derivation, the "<" operator would return the type NB, since each occurrence of the parent type, BOOLEAN, would be replaced by the derived type, NB. This is illustrated by the operators implicitly declared for type NNB:

```

function "and" (L, R : NNB) return NNB;      -- predefined operator
function "<" (L, R : NNB) return BOOLEAN;    -- predefined operator
function "<" (L, R : NNB) return NNB;      -- derived subprogram

```

The first two operators shown above are implicitly declared because of RM 3.4/6 and the rules defining what operators are predefined for boolean types. The third operator is an implicitly declared derived subprogram, since the explicit declaration of "<" in package P makes this subprogram derivable (RM 3.4/11). As part of the derivation process, each use of the parent type in the explicit declaration of "<" is replaced by a use of the derived type, so "<" returns a value of type NNB.

S8. Implicitly declared subprograms can, of course, be named explicitly. For example, one can write Q."and" and Q."<".

S9. RM 3.4/6 says that for every predefined operator of the parent type there is a corresponding operation for the derived type. This rule needs to be interpreted carefully:

```

type FLT is digits 5;
-- has predefined operator FLT ** INTEGER
type NT is new INTEGER;
-- no FLT ** NT operator is declared

```

The rule does not mean that every visible predefined operator having an operand of the parent type, INTEGER, is implicitly declared for NT. It means that NT has the same set of predefined operators as does INTEGER. In particular, it has the operator NT ** INTEGER.

S10. If a default expression exists for a component of the parent type, the same expression exists for the derived type, and the expression has the same meaning it had for the parent type, i.e., names of objects in the expression still denote the same entities:

```

package P is
  X : INTEGER := 5;
  type REC is
    record
      C : INTEGER := X;
    end record;
end P;

package Q is
  X : INTEGER := 6;
  type NEW_REC is new P.REC;
  RVAR : NEW_REC;
end Q;

```

RVAR.C has the value 5, not the value 6.

S11. Default expressions for subprogram parameters are treated in the same way when the subprogram is derived — the entities denoted by the names in the default expression do not change when the subprogram is derived.

S12. If an aspect of a parent type's representation has been specified by an implicit or explicit representation clause, and no explicit representation clause is given for the same aspect of the derived type, the representation of the parent and derived types are the same with respect to this aspect (RM 3.4/10 and AI-00138). For example,

```

type A is (A1, A2, A3);
for A'SIZE use 3;
type B is new A;
type C is new B;

```

A'SIZE = B'SIZE = C'SIZE = 3. A different size specification could be given for type B or type C (AI-00138).

S13. Implicitly declared subprograms can be hidden by an explicitly declared subprogram, but the extent to which the implicitly declared program is hidden depends on where the explicit declaration appears. For example:

```

package P is
  type T is range 1..10;
  function F return T;
end P;

with P;
pragma ELABORATE (P);
package Q is
  type NT is new P.T;
  -- implicit declaration of F occurs here
  X : NT := F;           -- denotes implicitly declared F
private
  function F return NT;  -- hides implicit declaration of F
end Q;

```

```

with Q;
package R is
    type NNT is new Q.NT;
    -- F implicitly declared here
end R;

```

The explicit declaration of F hides the derived subprogram F within Q's package body and private part, but the implicitly declared F is visible outside of package Q and prior to F's explicit declaration (RM 8.3/17). Thus, the F appearing in X's declaration denotes the implicitly declared F (whose body has been elaborated, so it can be invoked without raising PROGRAM_ERROR (see RM 3.9/5, 8)). Moreover, F's declaration in the visible part of Q means that it is a derivable subprogram of the second kind (RM 3.4/11). Thus R.NNT derives the function that was originally declared in package P.

S14. It is possible for a derived subprogram to be hidden completely throughout its scope:

```

with P;          -- the P declared in the previous example
package Q1 is
    type NT1 is private;
    function F return NT1;
private
    type NT1 is new P.T;
    -- implicit declaration of F
end Q1;

```

The derived F is hidden by the explicit declaration that occurs prior to the derived type declaration.

S15. A derived subprogram is not further derivable if an explicitly declared subprogram with the same profile and designator is given in a visible part:

```

package P is
    type T is range 1..10;
    function F return T;
end P;

with P;
pragma ELABORATE (P);
package Q is
    type NT is new P.T;
    -- derived F implicitly declared here
    X : NT := F;          -- uses derived F

    function F return NT;
    type REC is
        record
            C : NT := F;    -- uses explicitly declared F
        end record;
end Q;

with Q;
package R is
    type NNT is new Q.NT;
    -- derives explicitly declared Q.F;
end R;

```

Subprogram P.F is derivable and a corresponding subprogram is implicitly declared for type NT in package Q. The derived subprogram is visible until the explicit declaration of F occurs, so the derived F is called to initialize Q.X, and the new F is used to initialize component C of the record type. The explicit declaration of F in Q means the implicitly declared F is no longer derivable (RM 3.4/11). Thus NNT derives the function explicitly declared in Q rather than the implicitly declared function. (In addition, the name Q.F refers to the explicitly declared F; RM 8.3/7.)

S16. A type derivation can derive two subprograms with identical profiles (Ai-00012):

```

package P is
  type T is private;

  package Q is
    type U is range 1..10;
    procedure PROC (X1 : T; Y1 : U);
    procedure PROC (X2 : U; Y2 : T);
  end Q;
private
  type T is new Q.U;           -- legal (AI-00012)
end P;

```

Even though the subprograms derived for T are homographs, PROC (X1 : T; Y1 : T) and PROC (X2 : T; Y2 : T), the type derivation is legal (AI-00012).

S17. A type derivation is not illegal if a derived subprogram has the same identifier as a non-overloadable entity previously declared in the same declarative region:

```

package P is
  type T is range 1..10;
  function F return T;
end P;

package Q is
  F : INTEGER;
  type NT is new P.T;       -- derived F is hidden
end Q;

```

S18 Subprograms are not derivable until after the end of the visible part. For example:

```

package P is
  type T is (E1, E2);
  function F return T;
  type NEW_T is new T;
  function "<" (L, R : T) return BOOLEAN;
end P;

```

The function F is not derived for NEW_T since the type derivation occurs prior to the end of the visible part. In addition, the explicitly declared "<" for T is not derived for NEW_T. NEW_T only has the implicitly declared predefined "<" operator.

S19. When the parent type is declared in the visible part of a package specification, only those subprograms declared in the visible part where a parent type is declared can be derived. For example, consider:


```

package P is
  type T is new INTEGER;
  procedure PP(X : T);
end P;

with P;
package Q is
  procedure RR(X : P.T);
end Q;

with P, Q;
package S is
  procedure SS(X : P.T);
  type U is new P.T;
end S;

```

Q.RR is not derived by the declaration of U. Only P.PP is derived.

S20. If the parent type is not declared in the visible part of a package specification, then no subprograms are derived except those (if any) that were derived for the parent type. For example, using package P from the previous example:

```

with package P;
procedure R is
  type NT is new P.T;      -- derives procedure PP
  procedure R1 (X : NT) is ... end R1;
begin
  declare
    type NNT is new NT; -- derives only procedure PP

```

If procedure R1 had been given the name PP, this newly named procedure would not be derived for NNT; instead, NNT would derive the same procedure PP as before, even though this procedure is now hidden by the explicit declaration of the renamed R1.

S21. RM 3.4/12 says that among the implicitly declared subprograms for a derived type, "the implicit declarations of any derived subprograms occur last." The fact that these subprogram declarations occur last has no detectable semantic consequences.

S22. For purposes of understanding the semantic effect of calling a derived subprogram, RM 3.4/14 states that each actual parameter is replaced by a type conversion of the actual parameter to the parent type. The existence of such conversions does not affect overloading resolution:

```

package P is
  type T is array (1..10) of CHARACTER;
  function F (X : T) return T;
end P;

package Q is
  type NT is new P.T;
  Y : NT := Q.F("ABC");      -- unambiguous
  Z : NT := NT(P.F(P.T("ABC"))); -- ambiguous conversion operand
end Q;

```

The second call is ambiguous because the type of "ABC" cannot be determined from the context; "ABC" could have type STRING, T, or NT since string literals for any of these types can be written within Q. The fact that the first call is semantically equivalent to:

```
NT(P.F(P.T("ABC")))
```

is irrelevant.

S23. Care must be taken in determining the subtypes associated with parameters of derived subprograms. For example:

```
package P is
  type MY_INT is range 1..100;
  subtype MY_INT_50 is MY_INT range 1..50;
  subtype MY_INT_51 is MY_INT range 51..100;

  A : MY_INT;

  function "+" (L, R : MY_INT) return MY_INT;
  function G (X : MY_INT_50) return MY_INT_51;
end P;
```

A + 0 raises CONSTRAINT_ERROR for the new "+" operator because 0 is not in the range of MY_INT values and "+" has been redefined to accept only MY_INT arguments. The new declaration of "+" is semantically equivalent to:

```
function "+" (L, R : MY_INT'BASE range 1..100)    -- illegal notation
  return MY_INT'BASE range 1..100;                -- illegal notation
```

(The notation MY_INT'BASE range 1..100 cannot actually be written, since MY_INT'BASE is not a legal type mark; moreover, the formal parameter and return type must be specified with a type mark, not a subtype indication. However, this notation suggests the semantic effect of the actual declaration in a way that is particularly useful when we next discuss the effect of deriving this subprogram.)

S24. Now suppose we define a new package:

```
with P;
package R is
  type NEWER is new P.MY_INT range 51..90;
  RD : NEWER := 90;
end R;
```

The derived "+" subprogram uses the base type for NEWER and the range constraint declared for the original "+" operation:

```
function "+" (L, R : NEWER'BASE range 1..100)
  return NEWER'BASE range 1..100;
```

Hence, RD + 2 will not raise CONSTRAINT_ERROR even though 2 is not in the range of NEWER values. Moreover, if the redefined "+" implements the usual rules for addition, CONSTRAINT_ERROR will not be raised by RD + 2 even though the value returned lies outside the range of NEWER values, since the result is implicitly converted to the derived (i.e., base) type NEWER'BASE, which has the range of the parent type, MY_INT'BASE, and MY_INT'BASE has a range that includes the values 1..100. CONSTRAINT_ERROR will only be raised if the result returned by "+" lies outside the range permitted by the original operation, P."+". For example CONSTRAINT_ERROR would be raised by RD + 14, because this yields a value greater than 100.

Applying the same rules to G, we get:

```

function G(X : NEWER'BASE range 1..50)
    return NEWER'BASE range 51..100;

```

In this case, every call to G with a NEWER variable will raise CONSTRAINT_ERROR, since no NEWER value lies in the range 1..50. However, a call with a literal, e.g., G(5) would be acceptable, since the literal is converted to NEWER's base type (RM 4.6/15).

S25. The parent type in a derived type definition cannot, in some circumstances, be a derived type (RM 3.4/15). According to RM 3.4/1, a derived type definition is the only means of producing something called a derived type. In particular, an integer type definition does not create a derived type, since an integer type definition does not syntactically include a derived type definition:

```

package P is
    type T is range 1..10;
    type NT is new T;           -- legal
    type NNT is new NT;        -- illegal; NT is a derived type
end P;

```

The parent type T is not a derived type despite the equivalence stated in RM 3.5.4/5, since no derived type definition appears in the declaration for T.

Approved Interpretations

S26. If an aspect of a parent type's representation has been specified by an implicit or explicit representation clause and no explicit representation clause is given for the same aspect of the derived type, the representation of the derived and parent types are the same with respect to this aspect (AI-00138).

S27. An explicit length clause for STORAGE_SIZE of a task type, for SIZE (of any type), or for SMALL of a fixed point type, an explicit enumeration representation clause, an explicit record representation clause, or an explicit address clause for a task type can be given for a derived type (prior to a forcing occurrence for the type) even if a representation clause has also been given (explicitly or implicitly) for the same aspect of the parent type's representation (AI-00138). (But only a length clause is allowed for a derived type if the parent type has derivable subprograms.)

S28. Two derived subprograms can be homographs (AI-00012).

Changes from July 1982

S29. Each derivable subprogram of the parent type is declared implicitly for the derived type.

S30. A derived type declared immediately within the visible part of a package cannot be used as the parent type of a derived type definition, within the same visible part.

Changes from July 1980

S31. For a derived boolean type, the predefined relational operators return a result of type BOOLEAN.

S32. A representation clause can be given for a parent type after it is used in a derived type definition.

Legality Rules

L1. If a derived type is declared immediately within the visible part of a package, then within this visible part, this type must not be used as the parent type of a derived type definition (RM 3.4/15).

Exception Conditions

The only exceptions are those raised by elaboration of a subtype indication (see IG 3.3.2/E).

Test Objectives and Design Guidelines

T1. Check that the required predefined operations (and only these predefined operations) are declared (implicitly) for derived enumeration types:

- excluding boolean types: the enumeration literals of the base type, assignment, membership tests, qualification, explicit conversion to and from the parent type, the relational operators (<, <=, =, /=, >=, and >), for prefixes denoting an enumeration type or subtype, the attributes BASE, FIRST, LAST, SIZE, WIDTH, POS, VAL, SUCC, PRED, IMAGE, and VALUE, and for prefixes denoting an object having an enumeration type, the attributes SIZE and ADDRESS.
- for derived boolean types: the enumeration literals TRUE and FALSE, assignment, membership tests, qualification, explicit conversion to and from the parent type, the relational operators (<, <=, =, /=, >=, and >) (check that these operators return values of type BOOLEAN), the logical operators (not, and, or, xor), the short-circuit control forms (and then, or else) (check that the logical operators and short-circuit control forms return values of the derived boolean type), for prefixes denoting a boolean type or subtype, the attributes BASE, FIRST, LAST, SIZE, WIDTH, POS, VAL, SUCC, PRED, IMAGE, and VALUE, and for prefixes denoting an object having a boolean type, the attributes SIZE and ADDRESS.

Implementation Guideline: Check that the attributes return values of the derived (base) type rather than the derived subtype, when appropriate.

Implementation Guideline: For attributes or operations that depend on the subtype, check that appropriate results are obtained even when the subtype is nonstatic.

Implementation Guideline: Check that no additional operators or attributes are declared for a derived type.

Check that all values of the parent (base) type are present for the derived (base) type when the derived type definition is constrained.

Check that any constraint imposed on the parent subtype is also imposed on the derived subtype.

Implementation Guideline: Use two cases: 1) the derived type definition contains an explicit constraint, and 2) the type mark in the derived type definition already is constrained.

T2. Check that the required predefined operations (and only these predefined operations) are declared (implicitly) for derived integer types: integer literals, assignment, membership tests, qualification, explicit conversion of integer and real types (including conversion to and from the parent type), implicit conversion from *universal_integer*, the relational operators (<, <=, =, /=, >=, and >), the arithmetic operators (unary +, -, and abs, binary +, -, *, /, **, mod, and rem), for prefixes denoting an integer type or subtype, the attributes BASE, FIRST, LAST, SIZE, WIDTH, POS, VAL, SUCC, PRED, IMAGE, and VALUE, and for prefixes denoting an object having an integer type, the attributes SIZE and ADDRESS.

Check that all values of the parent (base) type are present for the derived (base) type when the derived type definition is constrained.

Check that any constraint imposed on the parent subtype is also imposed on the derived subtype.

Implementation Guideline: Follow the implementation guidelines for T1.

- T3. Check that the required predefined operations (and only these predefined operations) are declared (implicitly) for derived floating point types: real literals, assignment, membership tests, qualification, explicit conversion from an integer, floating point, or fixed point type (including to and from the parent type), implicit conversion from *universal_real* values, relational operators (<, <=, =, /=, >=, and >), arithmetic operators (unary +, -, and abs, binary +, -, *, /, and **), for prefixes denoting a floating point type or subtype, the attributes BASE, FIRST, LAST, SIZE, DIGITS, MANTISSA, EPSILON, EMAX, SMALL, LARGE, SAFE_EMAX, SAFE_SMALL, SAFE_LARGE, MACHINE_RADIX, MACHINE_MANTISSA, MACHINE_EMAX, MACHINE_EMIN, MACHINE_ROUNDS, and MACHINE_OVERFLOW, and for prefixes denoting an object having a fixed point type, the attributes SIZE and ADDRESS.

Check that all values of the parent (base) type are present for the derived (base) type when the derived type definition is constrained.

Check that any constraint imposed on the parent subtype is also imposed on the derived subtype.

Implementation Guideline: Follow all implementation guidelines for T1.

- T4. Check that the required predefined operations (and only these predefined operations) are declared (implicitly) for derived fixed point types: real literals, assignment, membership tests, qualification, explicit conversion from an integer, floating point, or fixed point type (including to and from the parent type), implicit conversion from type *universal_real*, relational operators (<, <=, =, /=, >=, and >), arithmetic operators (unary +, -, and abs, binary +, -, *, and /), for prefixes denoting a fixed point type or subtype, the attributes BASE, FIRST, LAST, SIZE, DELTA, MANTISSA, SMALL, LARGE, FORE, AFT, SAFE_SMALL, SAFE_LARGE, MACHINE_ROUNDS, and MACHINE_OVERFLOW, and for prefixes denoting an object having a fixed point type, the attributes SIZE and ADDRESS.

Check that all values of the parent (base) type are present for the derived (base) type when the derived type definition is constrained.

Check that any constraint imposed on the parent subtype is also imposed on the derived subtype.

Implementation Guideline: Follow all implementation guidelines for T1.

- T5. Check that the required predefined operations (and only these predefined operations) are declared (implicitly) for derived array types. In particular:

- for derived one-dimensional array types whose component type is not a character or boolean type: assignment (except when the component type is limited), aggregates (except when the component type is limited), membership tests, indexed components, qualification, explicit conversion to and from the parent type, explicit conversion between the derived type and an array type whose index type is convertible to that of the derived type and whose component type is the same as that of the derived type, slices, equality and inequality (unless the component type is limited), catenation (unless the component type is limited), the relational operators (<, <=, >, and >=) if the component type is an enumeration or integer type, for prefixes denoting an array type or subtype, the attributes BASE and SIZE; for prefixes denoting a constrained array subtype, an array object, or an array value, the attributes FIRST, FIRST(1), LAST, LAST(1), RANGE, RANGE(1), and LENGTH, LENGTH(1); and for prefixes denoting an object, the attributes SIZE and ADDRESS.
- for derived one-dimensional array types whose component type is a character

type: assignment, aggregates, membership tests, indexed components, qualification, explicit conversion to and from the parent type, explicit conversion between the derived type and an array type whose index type is convertible to that of the derived type and whose component type is the same as that of the derived type, slices, string literals, equality and inequality, catenation, the relational operators (<, <=, >, and >=), for prefixes denoting an array type or subtype, the attributes BASE and SIZE; for prefixes denoting a constrained array subtype, an array object, or an array value, the attributes FIRST, FIRST(1), LAST, LAST(1), RANGE, RANGE(1), and LENGTH, LENGTH(1); and for prefixes denoting an object, the attributes SIZE and ADDRESS.

- for derived one-dimensional array types whose component type is a boolean type: assignment, aggregates, membership tests, indexed components, qualification, explicit conversion to and from the parent type, explicit conversion between the derived type and an array type whose index type is convertible to that of the derived type and whose component type is the same as that of the derived type, slices, equality and inequality, catenation, the relational operators (<, <=, >, and >=), the logical operators (not, and, or, and xor) returning the derived type, for prefixes denoting an array type or subtype, the attributes BASE and SIZE; for prefixes denoting a constrained array subtype, an array object, or an array value, the attributes FIRST, FIRST(1), LAST, LAST(1), RANGE, RANGE(1), and LENGTH, LENGTH(1); and for prefixes denoting an object, the attributes SIZE and ADDRESS.
- for derived multidimensional array types: assignment (except when the component type is limited), aggregates (except when the component type is limited), membership tests, indexed components, qualification, explicit conversion to and from the parent type, explicit conversion between the derived type and an array type whose index types are convertible to those of the derived type and whose component type is the same as that of the derived type, equality and inequality (unless the component type is limited); for prefixes denoting an array type or subtype, the attributes BASE and SIZE; for prefixes denoting a constrained array subtype, an array object, or an array value, the attributes FIRST, FIRST(N), LAST, LAST(N), RANGE, RANGE(N), and LENGTH, LENGTH(N); and for prefixes denoting an object, the attributes SIZE and ADDRESS. (Check that N in the attributes must not be less than 1 or greater than the number of dimensions, and must be a static *universal_integer* expression.)

Check that all values of the parent (base) type are present for the derived (base) type when the derived type definition is constrained.

Implementation Guideline: Define a CREATE function that returns a value of the derived type that does not belong to the derived subtype, and check that no exception is raised when the equality operator is used.

Check that any constraint imposed on the parent subtype is also imposed on the derived subtype.

Implementation Guideline: Follow all implementation guidelines for T1.

- T6. Check that the required predefined operations (and only these predefined operations) are declared (implicitly) for derived record types. In particular:
- for a derived record type without discriminants: assignment (if the parent type is not limited), aggregates (if the parent type is not limited), membership tests, selection of record components, qualification, conversion to and from the

parent type, the equality and inequality operators (if the parent type is not limited); for prefixes denoting a record type or subtype, the attributes BASE and SIZE; for prefixes denoting a record object, the attributes SIZE and ADDRESS; and for prefixes denoting a component of a record object, the attributes POSITION, FIRST_BIT, and LAST_BIT.

- for a derived record type with discriminants: assignment (if the parent type is not limited), aggregates (if the parent type is not limited), membership tests, selection of record components (including discriminants), qualification, conversion to and from the parent type, the equality and inequality operators (if the parent type is not limited); for prefixes denoting a record type or subtype, the attributes BASE and SIZE; for prefixes denoting a record object, the attributes SIZE, CONSTRAINED, and ADDRESS; and for prefixes denoting a component of a record object, the attributes POSITION, FIRST_BIT, and LAST_BIT.

Check that all values of the parent (base) type are present for the derived (base) type when the derived type definition is constrained.

Implementation Guideline: Define a CREATE function that returns a value of the derived type that does not belong to the derived subtype, and check that no exception is raised when the equality operator is used.

Check that any constraint imposed on the parent subtype is also imposed on the derived subtype.

Implementation Guideline: Follow all implementation guidelines for T1.

- T7. Check that the required predefined operations (and only these predefined operations) are declared (implicitly) for derived access types. In particular:

- for derived access types whose designated type is not an array type, a task type, a record type, or a type with discriminants: assignment, allocators, membership tests, qualification, explicit conversion to and from the parent type, the literal null, formation of a selected component with the selector all, the equality and inequality operations; for prefixes denoting an access type or subtype, the attributes BASE, SIZE, and STORAGE_SIZE; and for prefixes denoting an access object, the attributes SIZE and ADDRESS.
- for derived access types whose designated type is an array type: assignment, allocators, membership tests, qualification, explicit conversion to and from the parent type, the literal null, formation of a selected component with the selector all, the equality and inequality operations, formation of indexed components, formation of slices (if the designated type is a one-dimensional array type); for prefixes denoting an access type or subtype, the attributes BASE, SIZE, and STORAGE_SIZE; for prefixes denoting an access object or value, the attributes FIRST, FIRST(N), LAST, LAST(N), RANGE, RANGE(N), LENGTH, and LENGTH(N); and for prefixes denoting an access object, the attributes SIZE and ADDRESS. (Check that N in the attributes must not be less than 1 or greater than the number of dimensions, and must be a static *universal_integer* expression.)
- for derived access types whose designated type is a task type: assignment, allocators, membership tests, qualification, explicit conversion to and from the parent type, the literal null, formation of a selected component with the selector all, selection of entries and entry families of the designated task type, the equality and inequality operations; for prefixes denoting an access type or subtype, the attributes BASE, SIZE, and STORAGE_SIZE; for prefixes denoting an access object, the attributes SIZE and ADDRESS; and for

prefixes denoting an access object or value, the attributes TERMINATED and CALLABLE.

- for derived access types whose designated type is a record type (with or without discriminants) or a private type with discriminants: assignment, allocators, membership tests, qualification, explicit conversion to and from the parent type, the literal null, formation of a selected component with the selector all, selection of the components and discriminants of the designated type (only the discriminant, for a private type), the equality and inequality operations; for prefixes denoting an access type or subtype, the attributes BASE, SIZE, and STORAGE_SIZE; and for prefixes denoting an access object, the attributes SIZE and ADDRESS.

Check that all values of the parent (base) type are present for the derived (base) type when the derived type definition is constrained.

Implementation Guideline: Define a CREATE function that returns a value of the derived type that does not belong to the derived subtype, and check that no exception is raised when the equality operator is used.

Check that any constraint imposed on the parent subtype is also imposed on the derived subtype.

Implementation Guideline: Follow all implementation guidelines for T1.

- T8. Check that the required predefined operations (and only these predefined operations) are declared (implicitly) for derived task types: formation of calls to entries and entry families of the parent type; qualification; membership tests; explicit conversion to and from the parent type; for prefixes denoting a type, the attribute BASE; for prefixes denoting a task object or type, the attributes STORAGE_SIZE, SIZE, and ADDRESS; and for prefixes denoting a task object or value, the attributes CALLABLE and TERMINATED.
- T9. Check that the required predefined operations (and only these predefined operations) are declared (implicitly) for derived private types. In particular:
- for derived private types that are not limited: assignment, membership tests, qualification, explicit conversion (to and from the parent type), the equality and inequality operators, selection of discriminant components (if the parent type has discriminants); for prefixes denoting private types or subtypes, the attributes BASE, SIZE, and CONSTRAINED; for prefixes denoting an object, the attributes SIZE and ADDRESS; and for prefixes denoting an object with discriminants, the attribute CONSTRAINED.
 - for derived private types that are limited: membership tests, qualification, explicit conversion (to and from the parent type), selection of discriminant components (if the parent type has discriminants); for prefixes denoting private types or subtypes, the attributes BASE, SIZE, and CONSTRAINED; for prefixes denoting an object, the attributes SIZE and ADDRESS; and for prefixes denoting an object with discriminants, the attribute CONSTRAINED.

Check that all values of the parent (base) type are present for the derived (base) type when the derived type definition is constrained.

Implementation Guideline: Define a CREATE function that returns a value of the derived type that does not belong to the derived subtype, and check that no exception is raised when the equality operator is used.

Check that any constraint imposed on the parent subtype is also imposed on the derived subtype.

Implementation Guideline: Follow all implementation guidelines for T1.

- T11. Check that a derived type declaration is not considered exactly equivalent to an

anonymous declaration of the derived type followed by a subtype declaration of the derived subtype. In particular, check that a derived function cannot be used in the derived type declaration itself and that `CONSTANT_ERROR` can be raised when the subtype indication of the derived type declaration is elaborated (even though the constraint would satisfy the derived (base) type).

Implementation Guideline: Be sure to explicitly convert the derived function in the derived type declaration so the subtype indication has the correct types for its expression. Use a subtype indication that has a range constraint, index constraint, and discriminant constraint.

T12. Check that default expressions in derived record types and derived subprograms are evaluated using the entities denoted by the expressions in the parent type.

Implementation Guideline: The parent type should be a derived type as well as a nonderived type.

T13. Check that a representation clause can be given for a derived type whether or not the corresponding aspect of the parent's representation has been specified with an explicit representation clause (see IG 13.1.c/T4).

T14. Check that a derived subprogram is visible and further derivable under appropriate circumstances. In particular, check the following cases:

- the derived subprogram is implicitly declared in the visible part of a package and:
 - a subprogram is later declared explicitly (by a subprogram declaration, a renaming declaration, or a generic instantiation) in:
 - the same visible part: the derived subprogram is visible until the occurrence of the explicit redeclaration and then only the explicitly declared subprogram is visible; moreover, only the explicitly declared subprogram is further derivable.
 - in the private part: the derived subprogram is visible until the occurrence of the explicit redeclaration and then only the explicitly declared subprogram is visible; the implicitly declared derived subprogram is further derivable.
 - in the package body: the derived subprogram is visible until the occurrence of the explicit redeclaration and then only the explicitly declared subprogram is visible; the implicitly declared derived subprogram is further derivable.
 - no subprogram is later declared explicitly that is a homograph of the derived subprogram: the derived subprogram is further derivable.
- the derived subprogram is declared (implicitly) in the private part after an explicit declaration (use all three forms of explicit declaration) of a subprogram having the same profile (this is only possible when the derived type definition is for the full declaration of a private type): the derived subprogram is hidden and not visible either in the private part or body, nor is it further derivable if the explicit declaration occurred in the visible part; it is derivable if the explicit declaration occurs in the private part.
- a derived subprogram is further derivable if the derived type declaration occurs in a declarative part, even if the derived subprogram is subsequently hidden by an explicit declaration.

Check that the same effects occur for an implicitly declared predefined operator and an

explicitly declared operator homograph. That is, check that, in appropriate cases, the explicitly declared operator is derived and hides the implicitly declared operator for a derived type, and that the derived operator can be further derived in appropriate cases.

- T15. Check that a derived type declaration is allowed if the declaration derives two subprograms that are homographs (see IG 8.3/T9 and AI-00012).

Implementation Guideline: Check that if a derived subprogram has the same profile as an enumeration literal, the derived type declaration is legal, and the derived subprogram hides the enumeration literal (see IG 8.3/T32).

- T16. Check that a subprogram declared in the visible part of a package specification cannot be derived until the end of the visible part.

Implementation Guideline: Check that the subprogram is derivable in the private part or body of the package. Include a check when the package is a generic package and the parent type is itself derived from a generic formal type or a nongeneric type.

Check that if a type is declared outside the visible part of a package, and if subprograms are explicitly declared for this type, the subprograms are not derivable.

Implementation Guideline: Include types declared in a private part as well as types declared in a declarative part.

- T17. Check that a type declared with a derived type definition in the visible part of a package cannot be used as the parent type in a derived type declaration occurring later in the same visible part.

Implementation Guideline: The derived type should be an enumeration, integer, float, fixed, array, record, access, task, and private type. Include a case where the type is declared by a numeric type definition.

Check that a type declared in the visible part of a package with an enumeration type definition, integer type definition, real type definition, array type definition, record type definition, or access type definition, can be used as the parent type in a derived type definition occurring later in the same visible part. Similarly, check for a task type declaration.

- T18. Check that calls of derived subprograms are performed correctly, i.e., check that no call is ambiguous because of the implicit type conversions that must be performed, and that the constraints checked for parameter and result subtypes are the constraints of the parent subprogram, not the constraints of the derived subtype.

Implementation Guideline: Use named parameters in some calls.

3.5 Scalar Types

Semantic Ramifications

S1. The term "integer type" includes all predefined integer types, e.g., INTEGER, SHORT_INTEGER, LONG_INTEGER, etc. It also includes types derived from an integer type and the type *universal_integer*.

S2. The term "enumeration type" includes any user-defined or predefined enumeration type, as well as types derived from enumeration types.

S3. The term "real type" includes all the predefined floating point types, e.g., FLOAT, LONG_FLOAT, etc., as well as user-defined fixed and floating point types. It also includes the type *universal_real* (but not the type *universal_fixed*), types derived from real types, and the type DURATION.

S4. The attributes FIRST and LAST return values belonging to the base type of their prefix. This means no exception is raised when the prefix denotes a null range, e.g.:

```

subtype T is INTEGER range 1..0;
... T'FIRST ...           -- no exception

```

S5. A RANGE is a legal attribute (a range attribute) if A denotes a constrained array subtype, an array object, or an array value, or if A has an access type whose designated type is an array type (RM 3.6.2/2 and RM 3.8.2/2). Such an attribute defines a nonstatic range, since the prefix of the attribute is not a static scalar subtype (RM 4.9/8 and RM 4.9/11). This means the range attribute cannot be used in a context where a static range is required. Syntactically, a range is allowed in the following contexts:

- component_clause (RM 13.4/2)
- discrete_range (RM 3.6/2)
- range_constraint (RM 3.5/2)
- relation (RM 4.4/2)

The range used in a component clause must be static (RM 13.4/2). A discrete range used as the choice in a case statement or variant part must be static (RM 3.7.3/4 and RM 5.4/5); the choice in an aggregate must also be static if there is more than one component association (RM 4.3.2/3). Finally, the range of an integer type definition must be static (RM 3.5.4/3). Consequently, a range attribute cannot be used in these specific contexts.

S6. When the expressions in a range have a real type, any comparison of the bounds with real values is subject to the rules given in RM 4.5.7 (since the predefined relational operators are used (RM 3.5/3)). In particular, even if S'LAST < S'FIRST evaluates to TRUE (i.e., even if S appears to have a null range), there can be some value X (e.g., S'FIRST) such that X in S also evaluates to TRUE. The reason is that, if either S'FIRST or S'LAST are not model numbers and they have values within S'SMALL of each other, the result given by any relational operation is nondeterministic.

Changes from July 1982

S7. *There are no significant changes.*

Changes from July 1980

S8. A range attribute can be used as a range.

Legality Rules

- L1. The only attribute allowed as a range is the RANGE attribute (RM 3.5/2).
- L2. The simple expression in a range must have a scalar type (RM 3.5/3).
- L3. When a range constraint is used in a subtype indication, the type of the simple expressions (or the bounds of a range attribute) must be the same as the base type of the type mark of the subtype indication (RM 3.5/4).
- L4. When a range is used in a membership test, the type of the simple expressions (or the bounds of a range attribute) must be the same as the base type of the simple expression (RM 4.5.2/10).
- L5. When a range is used in the choice of an aggregate, the type of the simple expressions (or the bounds of a range attribute) must be the same as the type of the corresponding array index (RM 4.3.2/1).
- L6. When a range is used in the declaration of an entry family or an array type, or in a slice or loop parameter specification, the simple expressions in the range must have the same type (RM 3.6.1/2).

- L7. When a range constraint is used in an integer type definition, both bounds of the range must be static expressions having an integer type; the expressions need not have the same integer type (RM 3.5.4/3).
- L8. When a range constraint is used in a real type definition, both bounds of the range must be static expressions having a real type; the expressions need not have the same real type (RM 3.5.7/3).
- L9. A range attribute cannot be used in an integer type definition or a real type definition (it is not static; see IG 3.5.4/S) (RM 3.5.7/3 and RM 3.5.9/3).
- L10. If a range constraint is used in a subtype indication, the subtype indication must not appear in an allocator (RM 4.8/4).
- L11. The prefix of the attribute FIRST or LAST cannot be a record, an access, or a private type (RM 3.5/7, RM 3.6.2/2, and RM 3.8.2/2).

Exception Conditions

- E1. If a range constraint is used in a subtype indication, either directly or as part of a floating or fixed point constraint, CONSTRAINT_ERROR is raised if the range is not null and at least one of the bounds does not belong to the subtype denoted by the type mark of the subtype indication (RM 3.5/4 and RM 3.3.2/9).

Test Objectives and Design Guidelines

- T1. Check that in each context requiring the syntactic construct "range", a range attribute can be used except that when a static range is required or the range has the wrong type, the range attribute is illegal.

Implementation Guideline: The legal contexts are: in an enumeration or integer subtype indication of an object declaration or component subtype definition (see IG 3.6.2/T4), in a membership test (see IG 4.5.2.a/T2 and IG 4.5.2.d/T31), as a nonstatic choice in an aggregate (see IG 4.3.2/T24), in the declaration of an entry family (see IG 9.5/T41), in an index constraint (used in the declaration of an array type or subtype) (see IG 3.6.1.b/T80), in a slice (see IG 4.1.2/T7), and in a loop parameter specification (see IG 5.5.b/T6).

Implementation Guideline: The illegal contexts for a range attribute are: in a component clause (see IG 13.4/T11), as a choice in a case statement (see IG 5.4.a/T21) or a variant part (see IG 3.7.3/T3), as a choice in an aggregate having more than one component association (see IG 4.3.2/T1), and as the range in an integer type definition (see IG 3.5.4/T1).

- T2. Check that the types of the bounds in a range must be the same (see IG 3.6.1.a/T2).
- T3. Check that CONSTRAINT_ERROR is raised for a subtype indication when the lower or upper bound of a non-null range lies outside the range of the type mark.

Implementation Guideline: Check for enumeration, integer, fixed, and floating types.

Implementation Guideline: Include a subtype indication that uses a range attribute.

Check that no exception is raised for a null range as long as the bounds lie within the range of the base type.

Implementation Guideline: Check for enumeration, integer, fixed, and floating types.

Implementation Guideline: Repeat both checks when the type being constrained is a generic formal type.

- T4. Check that the prefix of FIRST and LAST cannot be a record, an access, or a private type (including a generic formal private type).

Implementation Guideline: The designated type for the access type should be a constrained array type.

- T5. Check that the value returned for T'FIRST and T'LAST need not belong to subtype T (see IG 3.5.5/T2, /T3, /T7, and /T8; IG 3.5.8/T1; and IG 3.5.10/T7).

3.5.1 Enumeration Types

Semantic Ramifications

S1. The operations declared for an enumeration type are:

```

basic operations
  assignment (RM 3.5.5/1)
  membership tests (RM 3.5.5/1)
  qualification (RM 3.5.5/1)
  conversion (the identity conversion) (RM 4.6/4)
operators
  relational operators (RM 3.5.5/15)
functions
  the enumeration literals themselves (RM 3.5.5/15)
attributes
  ADDRESS (RM 13.7.2/3)
  BASE (RM 3.3.3/9)
  FIRST (RM 3.5/8)
  LAST (RM 3.5/9)
  SIZE (RM 13.7.2/4)
  WIDTH (RM 3.5.5/4)
  POS (RM 3.5.5/6)
  VAL (RM 3.5.5/7)
  SUCC (RM 3.5.5/8)
  PRED (RM 3.5.5/9)
  IMAGE (RM 3.5.5/11)
  VALUE (RM 3.5.5/13)

```

S2. Character literals are implicitly declared as functions, even though such a declaration cannot be given explicitly. In particular, they can be passed as actual parameters to generic function parameters or be renamed:

```
function A return CHARACTER renames 'A';
```

S3. Two enumeration literals are overloaded (RM 3.5.1/5) even if their scopes do not overlap. For example, two enumeration literals declared in parallel blocks are overloaded if they have the same identifier. This "overloading," however, causes no overloading resolution difficulties, since the visibility rules suffice to disambiguate overloaded literals whose scopes do not overlap.

S4. An enumeration literal appearing in an enumeration type definition is considered an explicitly declared operation for purposes of the visibility rules (AI-00330). Because, from a visibility viewpoint, an enumeration literal is explicitly declared, an explicit declaration of a homograph is illegal (AI-00330):

```

package P is
  type ENUM is (A, B, C);
  X : ENUM := A;           -- enumeration literal A

  function A return ENUM;  -- illegal
  type B is range 1..10;   -- illegal
end P;

```

S5. Although an enumeration literal is a declared function, an implementation must remember that such functions were declared by an enumeration literal specification, because enumeration literals can be used in static expressions, but user-defined functions cannot (RM 4.9/7).

S6. When an enumeration type is the parent type in a derived type declaration, an implementation must distinguish derived enumeration literals from derived subprograms:

```
package P is
  type ENUM is (RED);
  type PRIV is private;
  function RED return PRIV;
private
  type PRIV is new ENUM;
  type NT is new PRIV;
  -- literal RED and function RED declared here
end P;
```

The implicitly declared derived subprogram RED hides the implicitly declared enumeration literal RED, since the enumeration literal is considered a predefined operation (RM 8.3/17 and AI-00002).

S7. RM 3.5.1/3 requires that the identifiers given in an enumeration type definition be distinct. RM 2.3/3 says, "Identifiers differing only in the use of corresponding upper and lower case letters are considered the same." This means case is ignored in deciding whether two identifiers in an enumeration type definition are distinct:

```
type E is (ALPHA, alpha); -- illegal
```

Case is, however, significant for character literals, so 'a', and 'A' are considered distinct.

S8. The IMAGE attribute does not preserve the case of an enumeration identifier as given in source code. For example:

```
type COLOR is (Red, Green, Blue, Yellow);
```

COLOR'IMAGE(Red) equals "RED" (RM 3.5.5/11).

Approved Interpretations

S9. If an enumeration literal is declared with an enumeration type definition, then a function having the same identifier as the enumeration literal and the same parameter and result type profile cannot also be declared immediately within the same declarative region. Similarly, a nonoverloadable declaration of the enumeration literal's identifier is not allowed immediately within the declarative region containing the enumeration type definition (AI-00330).

S10. If the implicit declaration of a derived enumeration literal is a homograph of the implicit declaration of a derived subprogram and these declarations occur immediately within the same declarative region, then the derived enumeration literal (like a predefined operation) is hidden by the other homograph. The derived enumeration literal is hidden within the entire scope of the derived subprogram's declaration (AI-00002).

Changes from July 1982

S11. A character literal is also considered the declaration of a parameterless function.

Changes from July 1980

S12. Enumeration literals are declared implicitly as parameterless functions.

Legality Rules

L1. The identifiers and character literals listed in an enumeration type definition must be distinct from each other (RM 3.5.1/3). (For identifiers, differences in case are ignored; RM 2.3/3.)

- L2. An identifier declared by an enumeration type definition must be distinct from any nonoverloadable identifier declared in the same declarative region and must be distinct from any explicitly declared function having the same parameter and result type profile (RM 8.3/17 and AI-00330).

Test Objectives and Design Guidelines

- T1. Check that at least one enumeration literal is required (either an identifier or a character literal), and that exactly one is permitted.

Check that enumeration literals can have the maximum length permitted for identifiers (see IG 2.3/T3).

- T2. Check that an enumeration literal belonging to one enumeration type may be declared in another enumeration type definition in the same declarative region.

Implementation Guideline: Use all forms of declarative region.

Check that a function homograph cannot be declared explicitly in the same declarative region as an enumeration literal (see IG 6.6/T1 and IG 8.3/T1-T8).

- T3. Check that duplicate enumeration literals (including character literals) are not permitted in a single enumeration type definition.

Implementation Guideline: Check that differences in case are ignored for identifiers but not for character literals.

- T5. Check that an enumeration literal (including a character literal) is considered a function by renaming it as a function (see IG 8.5/T19) and by passing it to a formal generic function parameter (see IG 12.3.6/T3).

- T6. Check that an enumeration type can have more than 256 enumeration literals.

3.5.2 Character Types

Semantic Ramifications

- S1. The operations declared for a character type are the same as those declared for any enumeration type:

basic operations

- assignment (RM 3.5.5/1)
- membership tests (RM 3.5.5/1)
- qualification (RM 3.5.5/1)
- conversion (the identity conversion) (RM 4.6/4)

operators

- relational operators (RM 3.5.5/15)

functions

- the enumeration literals themselves (RM 3.5.5/15)

attributes

- ADDRESS (RM 13.7.2/3)
- BASE (RM 3.3.3/9)
- FIRST (RM 3.5/8)
- LAST (RM 3.5/9)
- SIZE (RM 13.7.2/4)
- WIDTH (RM 3.5.5/4)
- POS (RM 3.5.5/6)
- VAL (RM 3.5.5/7)

SUCC (RM 3.5.5/8)
 PRED (RM 3.5.5/9)
 IMAGE (RM 3.5.5/11)
 VALUE (RM 3.5.5/13)

Changes from July 1982

S2. There are no changes.

Changes from July 1980

S3. There are no significant changes.

Legality Rules

L1. A character literal must not appear more than once in the declaration of a character type (RM 3.5.1/3).

Test Objectives and Design Guidelines

T1. Check that an enumeration type containing a character literal is considered a character type (see IG 8.7.b/T27).

3.5.3 Boolean Type

Semantic Ramifications

S1. The operations declared for a boolean type are the same as those declared for any enumeration type plus short-circuit control forms and the logical operators:

basic operations

assignment (RM 3.5.5/1)
 membership tests (RM 3.5.5/1)
 short-circuit control forms (RM 3.5.5/1)
 qualification (RM 3.5.5/1)
 conversion (the identity conversion) (RM 4.6/4)

operators

relational operators (RM 3.5.5/15)
 and or xor not (RM 3.5.5/15)

functions

the enumeration literals themselves (RM 3.5.5/15)

attributes:

ADDRESS (RM 13.7.2/3)
 BASE (RM 3.3.3/9)
 FIRST (RM 3.5/8)
 LAST (RM 3.5/9)
 SIZE (RM 13.7.2/4)
 WIDTH (RM 3.5.5/4)
 POS (RM 3.5.5/6)
 VAL (RM 3.5.5/7)
 SUCC (RM 3.5.5/8)
 PRED (RM 3.5.5/9)
 IMAGE (RM 3.5.5/11)
 VALUE (RM 3.5.5/13)

S2. The declaration:

`type MY_BOOL is (FALSE, TRUE);`

is not a boolean type since it is not derived, directly or indirectly, from the predefined type `BOOLEAN`. In particular, values of type `MY_BOOL` cannot be used in contexts where a boolean value is required, e.g., in the condition of an if statement.

S3. The position number of the `BOOLEAN` value `TRUE` must be one and `BOOLEAN'POS(FALSE)` must equal zero. An implementation, however, is free to represent these enumeration literals any way it chooses. In particular, it might give `TRUE` the representation -1, and `FALSE` the representation zero. Of course, it would then have to take care to ensure that `FALSE` was considered less than `TRUE`, and that the position number of `TRUE` was one.

Changes from July 1982

S4. There are no changes.

Changes from July 1980

S5. A boolean type is defined as the predefined type `BOOLEAN` or as a type derived from a boolean type.

Test Objectives and Design Guidelines

T2. Check that a type having just the enumeration literals `FALSE` and `TRUE` is not considered a boolean type.

Implementation Guideline: Check that the type does not have boolean operations and its values cannot be used in conditional expressions.

3.5.4 Integer Types

Semantic Ramifications

S1. The operations declared for an integer type are:

- basic operations (RM 3.5.5/1)
 - assignment
 - membership tests
 - qualification
 - explicit conversion (from any numeric type)
 - implicit conversion (from *universal_integer*)
- operators (RM 3.5.5/15)
 - relational operators
 - binary + and -
 - binary * and /
 - binary mod rem ** abs
- attributes
 - `ADDRESS` (RM 13.7.2/3)
 - `BASE` (RM 3.3.3/9)
 - `FIRST` (RM 3.5/8)
 - `LAST` (RM 3.5/9)
 - `SIZE` (RM 13.7.2/4)
 - `WIDTH` (RM 3.5.5/4)
 - `POS` (RM 3.5.5/6)
 - `VAL` (RM 3.5.5/7)
 - `SUCC` (RM 3.5.5/8)

PRED (RM 3.5.5/9)
 IMAGE (RM 3.5.5/10)
 VALUE (RM 3.5.5/12)

S2. Since the base type INTEGER exists, the following declaration must be accepted:

```
type U is range INTEGER'FIRST .. INTEGER'LAST;
```

However, it is not required that U's base type be INTEGER; an implementation might choose to represent U with the LONG_INTEGER base type if LONG_INTEGER is supported.

S3. A length clause given for a numeric type can affect the space used for stored values, but does not affect the values of the base type. For example:

```
type T is range 0..(2**8)-1;  
for T'SIZE use 8;
```

If this representation clause is accepted, stored values of type T will occupy 8 bits. Nonetheless, T's base type must be a signed type such that -T'LAST is a representable value. Hence, if X is a variable having type T, the expression -1 = X cannot raise an exception; -1 is a value of T's base type. In short, representation clauses for integer types only affect how values are stored; computations using predefined operations still use the underlying base type, since the predefined operations are declared for the base type, not the subtype.

S4. Although a range constraint is allowed to contain a range attribute (RM 3.5/2), an integer type definition cannot contain a range attribute:

```
type T is range A'RANGE;      -- illegal
```

RM 3.5.4/3 requires that the bounds specified for an integer type definition be static, and attributes of an array object or type are never static (RM 4.9/8). Of course, the range attribute can be used in a subtype indication:

```
subtype ST is T range A'RANGE;  -- legal
```

The range in a subtype indication is not required to be static.

S5. The equivalence given in RM 3.5.4/4-6 is not to be taken literally. In particular, all subtypes declared with an integer type definition are static (AI-00023), even though the use of the conversion operation in RM 3.5.4/5 would suggest that such types are not static since an expression containing a conversion is not static (RM 4.9).

S6. AI-00387 recommends, as a nonbinding interpretation, that implementations raise CONSTRAINT_ERROR instead of NUMERIC_ERROR when the Standard requires NUMERIC_ERROR to be raised. (The effect of a nonbinding interpretation is to allow implementations to be validated if they follow either the Standard or the recommended interpretation; moreover, nonbinding interpretations of the Standard are likely to become binding in a future version of the Standard.) Because of the recommendation, ACVC tests will check whether or not NUMERIC_ERROR is raised when the Standard requires it, but will also allow CONSTRAINT_ERROR to be raised instead.

Approved Interpretations

S7. CONSTRAINT_ERROR can be raised in place of NUMERIC_ERROR (AI-00387).

S8. Subtypes declared by an integer type definition are static (AI-00023).

Changes from July 1982

S9. There are no significant changes.

Changes from July 1980

S10. `NUMERIC_ERROR` is raised by an implicit conversion instead of `CONSTRAINT_ERROR`.

Legality Rules

- L1. The bounds given in an integer type definition must each have some integer type, but they need not have the same integer type (RM 3.5.4/3).
- L2. Static expressions must be used to define the bounds of an integer type definition (RM 3.5.4/3). (In particular, a range attribute cannot be used.)
- L3. An integer type definition is illegal if there is no predefined integer type (other than *universal_integer*) that covers the specified range of values (RM 3.5.4/6).

Test Objectives and Design Guidelines

- T1. Check that the bounds in an integer type definition must be static.
Implementation Guideline: In particular, check that a range attribute is not allowed.
- T2. Check that the bounds of an integer type definition need not have the same integer type.
- T3. Check that an integer type is rejected if its upper bound exceeds `SYSTEM.MAX_INT` or if its lower bound is less than `SYSTEM.MIN_INT`.
Implementation Guideline: Check these bounds in separate type declarations. The specified range should cover only a few values.
- T4. Check that the predefined integer types are equivalent to their base types, e.g., `INTEGER'FIRST = INTEGER (INTEGER'BASE'FIRST)`.

3.5.5 Operations of Discrete Types

Semantic Ramifications

S1. `POS` is defined for integer types as well as enumeration types. In particular, the position number for a negative integer value is the value itself (RM 3.5.5/9), so position numbers can be negative.

S2. The position number of a value is not to be confused with its representation; e.g.:

```
type ENUM is (A, B);
for ENUM use (A => 1, B => 3);
-- ENUM'POS(A) = 0
-- ENUM'POS(B) = 1
```

S3. Because certain attributes are defined as functions, they can be renamed. A renaming declaration cannot be written, however, if an attribute returns or requires an argument of type *universal_integer* since the name of this type cannot be written. In addition, the RM describes `VAL` as a "special" function because its parameter can have any integer type, but no such function specification can actually be written. This means the only attributes that can be renamed as functions are `SUCC`, `PRED`, `IMAGE`, and `VALUE`. These are also the only ones that can be given as generic actual parameters.

S4. `VAL`'s argument can have any integer type, e.g.:

```
LONG_VAR : LONG_INTEGER := 5;
INT_VAR  : INTEGER      := INTEGER'VAL(LONG_VAR);
```

The initialization expression for `INT_VAR` is legal and does not raise an exception as long as the value of `LONG_VAR` is in the range of type `INTEGER`.

S5. The use of upper or lower case letters in VALUE's argument is irrelevant except when the argument is a character literal. For enumeration literals that are identifiers, case is ignored (see RM 2.3/3), and for integer, decimal, or based literals, case is similarly ignored (RM 2.4.1/3 and RM 2.4.2/3).

S6. An integer literal is not allowed to have a negative exponent (RM 2.4.1/4), so INTEGER'VALUE("0E-0") must raise CONSTRAINT_ERROR.

Approved Interpretations

S7. The lower bound of the IMAGE of an enumeration value is one (AI-00234).

S8. It has been recommended (AI-00239) (as a nonbinding interpretation) that the image of a nongraphic character be the sequence of letters given in italics in the declaration of type CHARACTER (RM C/13). (The image given for nongraphic characters will affect the value returned by the WIDTH attribute.)

Changes from July 1982

S9. The attribute WIDTH is defined explicitly to yield the value zero for a null subtype.

S10. The conditions for raising CONSTRAINT_ERROR for VAL are now stated correctly in terms of the parameter value rather than in terms of the result.

S11. The exception conditions for SUCC and PRED are rephrased in terms of T's base type instead of in terms of T's subtype.

S12. The argument of PRED, SUCC, IMAGE, and POS must have base type T (instead of subtype T).

S13. The result of T'VAL has the base type of T. (Similarly, for SUCC, PRED, and VALUE).

S14. The lower bound of the result of T'IMAGE is one.

Changes from July 1980

S15. The attribute WIDTH is added.

S16. The attributes IMAGE and VALUE are no longer defined for real types.

S17. The attribute POS returns a value of type *universal_integer*.

S18. The attribute T'VAL can no longer be renamed.

S19. The result of T'IMAGE starts with a leading blank if the value is non-negative.

S20. T'VALUE raises CONSTRAINT_ERROR (not DATA_ERROR) if its argument is ill-formed.

Legality Rules

L1. The short-circuit control forms are defined only for boolean types (RM 3.5.5/1).

L2. The arithmetic operators are not predefined for enumeration types (RM 3.5.5/15).

L3. The logical operators (and the not operator) are not predefined for numeric types or for enumeration types other than boolean types (RM 3.5.5/15).

L4. The prefix of the attributes WIDTH, POS, VAL, SUCC, PRED, IMAGE, and VALUE must have an enumeration or integer base type (RM 3.5.5/2).

L5. The base type of the prefix and base type of the argument of POS, SUCC, PRED, and IMAGE must be the same (RM 3.5.5/6,8-10).

L6. The argument of the attribute VALUE must have the base type STRING (RM 3.5.5/12).

L7. The argument of the attribute VAL must have an integer base type (RM 3.5.5/7).

Exception Conditions

- E1. The evaluation of T'SUCC raises CONSTRAINT_ERROR if its argument equals the last value in the base type of T (RM 3.5.5/8).
- E2. The evaluation of T'PRED raises CONSTRAINT_ERROR if its argument equals the first value in the base type of T (RM 3.5.5/9).
- E3. The evaluation of T'VAL(I), where I is an integer value, raises CONSTRAINT_ERROR if I does not equal a position number of a value in T's base type (RM 3.5.5/7).
- E4. The evaluation of T'VALUE(S), where S is a string value, raises CONSTRAINT_ERROR if
 - S is ill-formed, i.e.,
 - T is an enumeration type and S does not have the syntax of an identifier or a character literal (RM 3.5.5/13), or S is not the string produced by IMAGE for a nongraphic character (RM 3.5.5/11); or
 - T is an integer type and S does not have the syntax of an integer literal, optionally preceded by a single plus or minus sign (RM 3.5.5/13); or
 - the value represented by S does not belong to the base type of T (RM 3.5.5/13).

Test Objectives and Design Guidelines

- T1. Check that the prefix of WIDTH, POS, VAL, SUCC, PRED, IMAGE, and VALUE cannot be fixed or floating point types.
Implementation Guideline: The argument to POS, SUCC, PRED, and IMAGE should have the same type as the prefix.
- T2. For an enumeration type other than a boolean or character type, check the results and type produced by the following attributes:
 -
 - WIDTH
 - yields the number of characters in the longest enumeration literal belonging to the prefix subtype (not the prefix base type).
 - yields zero when the subtype has a null range.
 - IMAGE
 - yields an enumeration literal all in upper case, regardless of the case used in the enumeration literal specification; no leading or trailing blanks.
 - yields a string whose lower bound is one (even when the enumeration literal is not the first literal in the enumeration literal specification).
 - VALUE
 - yields the correct value:
 - regardless of the use of upper or lower case in the input string.

- regardless of the presence of leading or trailing blanks.
- when underline characters are the only characters distinguishing two enumeration values.
- even when the string contains the longest possible enumeration literal.
- can yield a result that need not belong to the subtype of the prefix.
- raises `CONSTRAINT_ERROR` if there is no corresponding value in the enumeration's base type.
- raises `CONSTRAINT_ERROR` if the string contains a leading or trailing horizontal tabulation character or another nongraphic character.
- raises `CONSTRAINT_ERROR` if the string is syntactically ill-formed:
 - it contains consecutive underscores, or a leading or trailing underscore.
 - the first character is a digit.
 - the string contains a blank between two identifiers.
Implementation Guideline: The identifiers should denote enumeration literals.
 - the string contains a graphic character that is not a letter or a digit.
- **PRED and SUCC**
 - the result need not be in the subtype of the prefix.
Implementation Guideline: Use a loop to check that each value is produced.
Implementation Guideline: Include a check for a type whose representation has been specified by a representation clause.
 - `CONSTRAINT_ERROR` is raised appropriately (see IG 3.5.5/T5).
- **POS**
 - the correct value is produced for each literal belonging to the base type of the prefix.
Implementation Guideline: Include a check for a type whose representation has been specified by a representation clause.
- **VAL**
 - the correct value is produced even when the enumeration literal corresponding to the value is hidden (e.g., by a user-defined function).
Implementation Guideline: Include a check for a type whose representation has been specified by a representation clause.
 - `CONSTRAINT_ERROR` is raised if the argument is negative or greater than or equal to the number of enumeration literals in the base type of the prefix.
- **FIRST and LAST**
 - yield correct values for the subtype denoted by the prefix.
Implementation Guideline: Include a case where the prefix denotes a null subtype.

Implementation Guideline: Repeat the tests for the case where the prefix is a generic formal discrete type (or a subtype of such a type; the subtype range constraint can be written using the VAL or VALUE attribute).

T3. For an integer type, check the results produced by the following attributes:

- **WIDTH**

- yields the number of characters in the decimal representation of one of the bounds of the prefix subtype (whichever bound yields the higher value).
- yields zero when the prefix has a null range.

- **IMAGE**

- yields the decimal representation of the argument, with a preceding blank or minus sign.
- yields a string whose lower bound is one.

- **VALUE**

- yields the correct value:
 - for decimal and based literals, with and without exponents.
 - regardless of the presence of leading or trailing blanks, or leading zeroes.
 - underline characters embedded in the literal are ignored.
- yields a result that need not belong to the subtype of the prefix.
Implementation Guideline: Include a string whose conversion might be expected to raise **NUMERIC_ERROR**.
- raises **CONSTRAINT_ERROR** if there is no corresponding value belonging to the base type.
- raises **CONSTRAINT_ERROR** if the string contains a leading or trailing horizontal tabulation character.
- raises **CONSTRAINT_ERROR** if the string is syntactically ill-formed:
 - the string contains consecutive underscores, or a leading or trailing underscore.
 - the string contains an underscore preceding or following the "E" or "e" character of the exponent.
 - the string contains an underscore in incorrect places for a based literal.
 - a negative exponent value is specified.
 - the number is terminated with a decimal point.
 - for a based literal, the extended digits are not all within the correct range for the number's base.
 - for a based literal, the base is less than 2 or greater than 16.

- PRED and SUCC

- the result need not be in the subtype of the prefix.
Implementation Guideline: Use a loop to check that each value is produced.
- CONSTRAINT_ERROR is raised appropriately (see T5).

- POS

- a negative value is produced for a negative argument value.

- VAL

- the correct result is produced for a negative position number as well as a positive one.
- the argument need not have the type of the prefix.
- CONSTRAINT_ERROR is raised if the argument lies outside the range of the prefix's base type.

- FIRST and LAST

- yield correct values for the subtype denoted by the prefix.
Implementation Guideline: Include a case where the prefix denotes a null subtype.

Implementation Guideline: Repeat the tests for the case where the prefix is a generic formal discrete type (or a subtype of such a type; the subtype range constraint can be written using the VAL or VALUE attribute).

- T4. Check that CONSTRAINT_ERROR is not raised when the argument to SUCC, PRED, POS, and IMAGE does not belong to the prefix subtype.

Implementation Guideline: Include a check when the prefix is a generic formal discrete or integer type.

- T5. Check that T'SUCC and T'PRED raise CONSTRAINT_ERROR when their arguments equal B'LAST and B'FIRST, respectively, where B is the base type of T.

Implementation Guideline: Check for both integer and enumeration types, including formal discrete and integer types.

- T6. Check that the argument to SUCC, PRED, IMAGE, and POS must have the base type of the prefix and cannot have a fixed or floating point type.

Implementation Guideline: Try at least one test using SUCC with a nondiscrete type, e.g., a fixed point type with a delta of 1.0.

Check that the argument to VALUE must have the base type STRING.

Implementation Guideline: Use an array of character type as the argument.

Check that the argument to VAL must be an integer type.

- T7. For a character type, check the results produced by the following attributes:

- WIDTH

- yields the number of characters in the longest enumeration literal belonging to the prefix subtype (not the prefix base type).

Implementation Guideline: Use an enumeration type in which a character literal is the longest literal.

- For the predefined type CHARACTER, check WIDTH for the image of a nongraphic character.

Implementation Guideline: AI-00239 recommends that nongraphic characters have an image such as NUL for CHARACTER'VAL(0), but this interpretation is nonbinding, so an

implementation might provide a different image, e.g., 'NUL' or '000'. In any event, the value of WIDTH for CHARACTER should reflect the representation chosen for nongraphic characters.

- IMAGE

- yields a character literal, preserving the case of the graphic character.

Implementation Guideline: Check the results for each graphic character. Include a check for a user-defined character type.

- yields a string whose lower bound is one.

- VALUE

- yields the correct value for an argument that is a character literal or the string representation of a nongraphic character.

Implementation Guideline: Check for all graphic and nongraphic characters.

- yields a result that need not belong to the subtype of the prefix.
 - raises CONSTRAINT_ERROR if there is no corresponding value in the prefix's base type.
 - raises CONSTRAINT_ERROR if the string contains a leading or trailing horizontal tabulation character or another nongraphic character.
 - raises CONSTRAINT_ERROR if the string is syntactically ill-formed, e.g., if the string is "", "''", "'a'", "a'", or 'ab'.

- PRED and SUCC

- the result need not be in the subtype of the prefix.

Implementation Guideline: Use a loop to check that each value is produced.

- CONSTRAINT_ERROR is raised appropriately (see IG 3.5.5/T5).

- POS

- the correct value is produced for each literal belonging to the base type of the prefix.

Implementation Guideline: Include tests for the nongraphic characters.

Implementation Guideline: Include a check for an enumeration type whose representation has been specified with a representation clause.

- VAL

- the correct value is produced even for nongraphic characters.
 - CONSTRAINT_ERROR is raised if the argument is negative or is greater than or equal to the number of enumeration literals in the base type of the prefix.

Implementation Guideline: Include a check for a type whose representation has been specified by a representation clause.

- FIRST and LAST

- yield correct values for the subtype denoted by the prefix.

Implementation Guideline: Include a case where the prefix denotes a null subtype.

Implementation Guideline: Repeat the tests for the case where the prefix is a generic formal discrete type (or a subtype of such a type; the subtype range constraint can be written using the VAL or VALUE attribute).

T8. For a boolean type, check the results produced by the following attributes:

-
- WIDTH
 - yields the correct value (4 or 5).
 - yields zero when the subtype has a null range.
- IMAGE
 - yields the value "TRUE" or "FALSE".
 - yields a string whose lower bound is one.
- VALUE
 - yields the correct value:
 - regardless of the use of upper or lower case in the input string.
 - regardless of the presence of leading or trailing blanks.
 - yields a result that need not belong to the subtype of the prefix.
 - raises **CONSTRAINT_ERROR** if there is no corresponding value in the enumeration's base type.
 - raises **CONSTRAINT_ERROR** if the string contains a leading or trailing horizontal tabulation character.
- PRED and SUCC
 - the result need not be in the subtype of the prefix.
 - **CONSTRAINT_ERROR** is raised appropriately (see IG 3.5.5/T5).
- POS
 - the correct value is produced for TRUE and FALSE.
- VAL
 - the correct value is produced.
 - **CONSTRAINT_ERROR** is raised if the argument is negative or greater than one.
- FIRST and LAST
 - yield correct values for the subtype denoted by the prefix.

Implementation Guideline: Include a case where the prefix denotes a null subtype.

Implementation Guideline: Repeat the tests for the case where the prefix is a generic formal discrete type (or a subtype of such a type; the subtype range constraint can be written using the VAL or VALUE attribute).

3.5.6 Real Types

Semantic Ramifications

S1. Although the RM gives prominence to the concept of model numbers, it is the *safe numbers that actually determine all the important computational properties of real types*. Safe numbers are discussed in IG 3.5.7/S.

S2. The concept of "model" floating and fixed point numbers is used to specify the minimum accuracy and range of values an implementation must allow for values of a given real type. An implementation is free to use a more accurate representation than that specified by a type definition. For example, a floating point type declaration might specify that floating point values are to be maintained with at least 18 bits of accuracy. Suppose a floating point type supported by the hardware provides 27 bits of accuracy. Since the hardware type provides more than the required accuracy, it can be used by the implementation; truncation to 18 bits is not required. Extra accuracy can also be provided if a machine performs floating point calculations in "overlength" registers. In this case, calculations might be performed with more bits of precision than is implied by the stored representation of values. In effect, different representations are provided for the same real type — a stored representation and a representation used for results of computations. If an optimizing compiler keeps results in registers, and these registers give more bits of precision than the stored values, then a comparison such as $A*B > C$ may return different values, depending on whether C is the result of another computation, and hence, is in a register, or is a stored value. Such uncertainties in computing with real values are reflected in the properties of Ada's model for real numbers, and hence, are permitted. But an implementation that takes advantage of the extra accuracy provided by a machine may surprise its users, so care should be taken.

S3. An algorithm written to depend just on the properties of the model numbers is completely portable. An algorithm that depends on the properties of the safe numbers is not portable, since the range of safe numbers may differ from one implementation to the next. However, the concept of safe numbers allows the RM to specify what the effect of real arithmetic operations must be for certain values that lie outside the range of model numbers.

S4. An algorithm can easily use more than the implementation-independent properties of a real type and hence be non-portable. A diagnostic compiler could issue warnings about such transgressions but when portability is not a requirement, it is not helpful to issue a warning. Moreover, the programmer can achieve portability by one of two means: first, by using only the guaranteed Ada properties of model numbers; second, by using machine parameters of the underlying hardware types. The Ada method is crude, but simple; on the other hand, the machine method can give a closer match to the actual hardware at the expense of some additional complexity.

Approved Interpretations

S5. `CONSTRAINT_ERROR` can be raised instead of `NUMERIC_ERROR` (AI-00387).

Changes from July 1982

S6. The set of safe numbers must include the set of model numbers for a type.

Changes from July 1980

S7. The notion of safe numbers has been introduced.

S8. Implicit conversion of a *universal_real* value raises `NUMERIC_ERROR` instead of `CONSTRAINT_ERROR`.

3.5.7 Floating Point Types

Semantic Ramifications

S1. The operations declared for a floating point type are:

```

    basic operations (RM 3.5.8/1)
        assignment
        membership tests
        qualification
        explicit conversion (from any numeric type)
        implicit conversion (from universal_real)
    operators (RM 3.5.8/15)
        relational operators
        unary + and -
        binary + and -
        * / ** abs
    attributes
        ADDRESS          (RM 13.7.2/3)
        BASE             (RM 3.3.3/9)
        FIRST            (RM 3.5/8)
        LAST             (RM 3.5/9)
        SIZE             (RM 13.7.2/4)
        DIGITS           (RM 3.5.8/4)
        MANTISSA         (RM 3.5.8/5)
        EPSILON          (RM 3.5.8/6)
        EMAX             (RM 3.5.8/7)
        SMALL            (RM 3.5.8/8)
        LARGE            (RM 3.5.8/9)
        SAFE_EMAX        (RM 3.5.8/11)
        SAFE_SMALL       (RM 3.5.8/12)
        SAFE_LARGE       (RM 3.5.8/13)
        MACHINE_ROUNDS   (RM 13.7.3/3)
        MACHINE_OVERFLOW (RM 13.7.3/4)
        MACHINE_RADIX    (RM 13.7.3/6)
        MACHINE_MANTISSA (RM 13.7.3/7)
        MACHINE_EMAX     (RM 13.7.3/8)
        MACHINE_EMIN     (RM 13.7.3/9)

```

S2. It is often believed that the RM's formula for computing the number of binary digits in the mantissa of model numbers is incorrect. The usual argument is that if D digits of accuracy are required, then it is sufficient if $2^{(-B)} \leq 10^{(-D)}$, i.e.:

$$\begin{aligned}
 -B * \log(2) &\leq -D * \log(10) \\
 B &\geq D * \log(10) / \log(2) \\
 B &= \text{ceiling}[D * \log(10) / \log(2)] \quad (\text{since } B \text{ is an integer})
 \end{aligned}$$

This gives a value one less than the value specified in RM 3.5.7/6. But the correct characterization of the requirement is that the largest relative error for model numbers must be less than the smallest relative error for the corresponding decimal numbers. (The goal is to guarantee that every D -digit decimal number can be represented as a unique model number, and such a model number can, in turn, be uniquely mapped back to the original decimal number.) To develop an appropriate formula based on the minimum and maximum relative error, recall that the relative error for a D -digit number is the place value of the least significant digit divided by the number. For example, the relative error of the 2-digit number 25 is $1/25$ (since 25 can represent any value between 24.5 and 25.5).

S3. The *maximum* relative error for a D-digit number is one divided by the smallest D-digit number, e.g., for $D = 2$, the maximum relative error is $1/10 = 10^{-(D+1)}$. The minimum relative error for a 2-digit decimal number is $1/9$ which is slightly larger than 10^{-D} , so a lower bound on the minimum relative error is 10^{-D} . Corresponding calculations apply to B-digit binary numbers. In short, in order to ensure that every decimal number has a unique model number representation, it is necessary that:

$$\begin{aligned} \max \text{ binary (model) number error} &\leq \min \text{ decimal number error, i.e.,} \\ 2^{-(B+1)} &\leq 10^{-D} \end{aligned}$$

This relation leads directly to the formula given in the RM. [A discussion of why model numbers can represent decimal numbers uniquely is given in I. B. Goldberg, "27 bits are not enough for 8-digit accuracy," *CACM* 10, 2 (Feb. 1967), pp. 105-106.]

S4. For a given specification of digits, the following table shows the minimum number of binary mantissa bits required and the minimum exponent range. (Note that $\log(10)/\log(2)$ is approximately 3.32.)

DIGITS	MANTISSA	EMAX	DIGITS	MANTISSA	EMAX
1	5	20	16	55	220
2	8	32	17	58	232
3	11	44	18	61	244
4	15	60	19	65	260
5	18	72	20	68	272
6	21	84	21	71	284
7	25	100	22	75	300
8	28	112	23	78	312
9	31	124	24	81	324
10	35	140	25	85	340
11	38	152	26	88	352
12	41	164	27	91	364
13	45	180	28	95	380
14	48	192	29	98	392
15	51	204	30	101	404

S5. The requirement that the exponent range be at least $-4^*B \dots 4^*B$ may restrict what hardware types can be used to represent a floating point type. For example, on the Honeywell 6000 in double precision, the maximum exponent value is $2.0^{**}127$, so the longest mantissa length allowed for model numbers is 31 bits or less. A 31-digit mantissa length corresponds to DIGITS = 9, which is approximately half the actual precision provided by the hardware type. Nonetheless, an implementation cannot use this hardware type to support a floating point type requiring DIGITS = 10 since the required exponent range is not supported. For most machines, the actual mantissa length of the hardware floating point types is the primary limitation restricting what range of DIGITS values can be accepted; in these cases, the set of model numbers guarantees a smaller exponent range than is actually provided. (The value 4^*B as EMAX was chosen because some minimal exponent range had to be guaranteed by a model that only used the value of digits to determine a model range, and 4^*B was thought to be a reasonable compromise between user needs and what most machines provide.)

S6. As another example, consider a floating point type that uses hexadecimal representation with the mantissa having six hexadecimal digits (24 binary digits) and a hexadecimal exponent range of -64 .. 63. The hexadecimal exponent range corresponds to the binary exponent range

-256 .. 252. However, since the first three bits of the first digit of the hexadecimal mantissa can be zero, the minimum number of mantissa bits in a binary representation is $24-3=21$. Moreover, for a binary mantissa, the exponent range must be adjusted to $-256-3 .. 252$, i.e., $-259 .. 252$. A 21-digit binary mantissa for a model number requires an exponent range of $-84 .. 84$, so it is clear that this hexadecimal floating point representation can be used as the base type for floating point types requiring no more than 6 decimal digits of accuracy. The safe numbers associated with this base type have an exponent range $-252 .. 252$, the largest symmetric exponent range supported by the hardware type.

S7. In considering the semantics of floating point types, it is helpful to keep the following points in mind. These are discussed more fully below.

- The safe numbers of a floating point subtype are the safe numbers of the base type (RM 3.5.7/9). The safe numbers of the base type are a subset of the values of the type. Model numbers of a type are a subset of the safe numbers.
- All floating point operations (except assignment) are performed using safe numbers.
- The model numbers of a type or subtype *T* are determined solely by the value of *T*'DIGITS. The model numbers serve to define a minimal set of safe numbers that must be supported when a digit's value is accepted (by an implementation) in a real type definition.

S8. The model numbers are a subset of the safe numbers because the mantissa length for a model number is less than or equal to the mantissa length for safe numbers; in addition, the exponent values for safe numbers can lie outside the range $-4*B .. 4*B$. The safe numbers are a subset of the values of a type because the mantissa length for a base type can exceed *B* (so values can be represented that fall between two safe numbers) and because values of the type can have exponent values that lie outside the range allowed for safe numbers.

S9. Arithmetic and relational operations are performed on safe numbers because these operations are declared just for a base type. RM 3.5.7(10-12) says that a floating point type declaration having the form:

```
type T is digits D range L .. R;
```

is semantically equivalent¹ to the following sequence of declarations:

```
type %FP is new predefined_floating_point_type;
subtype T is %FP digits D range %FP(L) .. %FP(R);
```

The only arithmetic and relational operators declared for type *T* are those declared for its base type %FP. Consequently, all arithmetic and relational operations are performed using the safe numbers of the base type. If *T*'BASE'DIGITS > *T*'DIGITS (i.e., if the base type has greater accuracy than is required by *T*'s declaration), then these operations will be performed with greater accuracy than is required for *T*. For example, consider:

```
type T is digits 3;
X : T := 126.0;
Y : T := 158.0;
```

¹This equivalence does not hold in all respects. In particular, it implies that *T* is a nonstatic type because conversions are not allowed in static expressions (RM 4.9), but AI-00023 says *T* is intended to be considered a static type.

126.0 and 158.0 are both model (and safe) numbers, i.e., they must be represented exactly as values of type T. Now consider $X * X$, which equals 15876.0. If `T'BASE'DIGITS` ≥ 5 , then 15876.0 is also a safe number, so the expression:

$$X * X = Y * 100.0 + 38.0 + 158.0$$

must evaluate to TRUE; every arithmetic operation in this expression uses and yields safe numbers, so the equality must be evaluated exactly (RM 4.5.7). The fact that the *model* numbers of subtype T have only 3 decimal digits of accuracy (RM 3.5.7/15) is not relevant here. If calculations were performed with only 3 digits of accuracy, the result of $Y * 100.0 + 38.0$ need not equal 15838.0, and the equality comparison could yield FALSE.

S10. RM 3.5.8/16 says:

The operations of a subtype are the corresponding operations of the type except for the following: assignment, membership tests, qualification, explicit conversion, and [certain attributes]; the effects of these operations are redefined in terms of the subtype.

What does redefining the effects of certain operations "in terms of the subtype" mean? Let us first consider the effects for conversion. Using the variables of the previous example and continuing to assume that `T'BASE'DIGITS` is 5 or greater, it must be the case that $X * X = 15876.0$ evaluates to true, since 15876.0 is implicitly converted to T's base type (RM 3.5.8/1), and 15876.0 is a safe number for the base type, as is the value of $X * X$. The same holds for an explicit conversion: $X * X = T(15876.0)$. The explicit conversion first converts 15876.0 to T's base type (RM 4.6/4) and then checks to see if the converted value belongs to subtype T; no rule allows the operand to be converted to a model number for T (i.e., a model number accurate to only 3 decimal digits). The only sense in which the effect of conversion to subtype T is "redefined" is that the converted value is checked against subtype T's range, not against the range of T's base type. (Such checks are performed using T's predefined operations (RM 3.5/3), i.e., using the safe numbers of the base type.) For example:

```
subtype ST is T range 15838.0 .. 15976.0;
```

The range, 15838.0 .. 15976.0 is equivalent to $T(15838.0) .. T(15976.0)$, and if the base type has at least 5 digits of accuracy, the bounds for ST are safe numbers. The conversion, $ST(15838.0)$, must not raise any exception, e.g., if 15838.0 were represented with three decimal digits of accuracy, it would have a mantissa of 11 bits, and would fall in the model interval 15832.0 .. 15840.0. It would be incorrect to compare 15838.0 with an incorrect lower bound such as 15840.0. (One can check that ST has the correct lower bound by evaluating $ST'FIRST = 15840.0 - 1.0 - 1.0$, which must evaluate to TRUE; if fewer than 5 digits of accuracy are being used, the result will not be determined exactly, and could evaluate to FALSE.)

S11. Membership tests and qualification are similarly defined in terms of comparisons using the predefined relational operators, and so must be evaluated using the safe numbers of the base type rather than the model numbers of a subtype.

S12. In short, the accuracy with which membership tests, qualification, and explicit conversion are performed is not affected by a subtype declaration. The only sense in which these operations are "redefined" in terms of a subtype is with respect to the range checks that must be performed.

S13. Assignment is a bit different. The intention is that values need only be stored with the accuracy specified for a subtype. For example, consider:

```
Z : ST; -- has bounds 15838.0 .. 15976.0
```

In particular, values stored in Z need only be represented with 3 digits of accuracy, even if the base type has more accuracy. This means a value being stored in Z can first be modified to a

3-digit model number, then checked against Z's bounds, and if the check shows that the modified number belongs to ST's range, the modified value can be stored in Z. It is important that the 3-digit approximation be done before the range check is made. For example, consider:

```
Z := 15838.0;
```

If 15838 is approximated to 11 bits of accuracy (3 decimal digits), it can be stored as the model number 15832.0, it is important to check the actual value being stored against ST's range; in this case, 15832.0 in ST'Range will yield FALSE, so CONSTRAINT_ERROR would be raised. If the value of the right-hand side were checked before being reduced to 3 digits of accuracy, then the check would find that 15838.0 does belong to ST's range, but the value assigned would be 15832.0, which is outside ST's range. Note that after such an approximation of a base type's value, Z = 15838.0 will evaluate to FALSE, since 15832.0 and 15838.0 are different model numbers when the base type has 5 digits of accuracy.

S14. From an implementation viewpoint, these considerations only mean that all operations for subtype T (with the possible exception of assignment) must be performed using operations of the base type. An implementation can safely round (or truncate) values to model numbers of the subtype only when executing an assignment statement. (Such an approximation is not allowed when temporary storage assignments are performed. For example, when evaluating $X*X + Y*Y$, if it is necessary to store one of the products, the product must be stored with the accuracy of the base type.)

S15. If a range constraint is given in a floating point type declaration, then the base type must be chosen not only to satisfy the specified value for DIGITS, but also so the bounds belong to the range of safe numbers (RM 3.5.7/12). This means that the value specified for DIGITS does not totally constrain the underlying base type:

```
type T is digits FLOAT'DIGITS range FLOAT'FIRST .. FLOAT'LAST;
```

This declaration might be rejected by an implementation because FLOAT'FIRST and FLOAT'LAST need not belong to the range of FLOAT's safe numbers and there need be no predefined type that includes the values of FLOAT'FIRST and FLOAT'LAST in its range of safe numbers.

S16. Real literals are converted implicitly to an appropriate real type (RM 4.6/15). If a literal is a safe number, it must be converted exactly (RM 4.5.7). If a literal is not a safe number, the converted value must lie between consecutive safe numbers (RM 4.5.7). Literals that are not safe numbers need not be converted consistently. For example, consider:

```
X := 0.1;
-- some calculation not changing X
if X = 0.1 then
```

The equality operation can yield FALSE because the representation of X can depend on whether a register value or a stored value is used. If a register has more mantissa bits than stored values, and if the literal 0.1 is loaded into the register with maximum precision, then the value representing 0.1 in the register can be different from the value stored in X.

S17. In a type definition of the form digits D range L .. R, L and R need not be expressions having the same floating point type, nor need they have a floating point type — L or R could be fixed point expressions.

S18. If an implementation supports SHORT_FLOAT and/or LONG_FLOAT, SHORT_FLOAT'DIGITS < FLOAT'DIGITS < LONG_FLOAT'DIGITS must hold. If an implementation does not support these precisions, then these identifiers must not be declared in STANDARD.

Changes from July 1982

S19. There are no significant changes.

Changes from July 1980

S20. The value of MANTISSA has been increased by 1.

S21. The notion of safe numbers has been introduced.

Legality Rules

- L1. The arithmetic operators `mod` and `rem` are not predefined for floating point types (RM 3.5.8/15).
- L2. The expression following `digits` must be a static expression having an integer type (RM 3.5.7/3).
- L3. In a floating point constraint, the expression following `digits` must have a value less than or equal to `SYSTEM.MAX_DIGITS` and greater than zero (RM 3.5.7/3 and RM 3.5.7/12).
- L3. If a range constraint is provided in a floating point constraint used in a real type definition, the bounds of the range constraint must be specified as static expressions having either a fixed point type, a floating point type, or the type *universal_real*; the expressions need not have the same real type (RM 3.5.7/3).
- L4. If a range constraint is provided in a floating point constraint used in a real type definition, the specified range must be included in the set of safe numbers defined for the base type (RM 3.5.7/12).
- L5. If a floating point constraint follows a type mark in a subtype indication, then the type mark must denote a floating point base type or subtype (RM 3.5.7/14), and the subtype indication must not appear in an allocator (RM 4.8/4).

Exception Conditions

- E1. If a floating accuracy definition appears in a subtype indication whose type mark is `T`, `CONSTRAINT_ERROR` is raised if the value of the expression following `digits` is greater than `T'DIGITS` (RM 3.5.7/14 and RM 3.3.2/9).
- E2. If a floating accuracy definition with a range constraint appears in a subtype indication whose type mark is `T`, `CONSTRAINT_ERROR` is raised if the specified range is not null and one (or both) of the bounds does not lie in the range `T'FIRST .. T'LAST` (RM 3.5.7/14, RM 3.5/3, and RM 3.3.2/9).

Test Objectives and Design Guidelines

- T1. Check that
 - a. the expression after `digits` must be an integer type.
 - b. the expression after `digits` must not be negative or zero.
 - c. the expression after `digits` must be static.
 - d. the expression after `digits` must have a value \leq `SYSTEM.MAX_DIGITS`.
 - e. the expressions in a floating point range constraint must not have an integer type.
 - f. if a range is specified, the upper and lower bounds must be included in the range of safe numbers.

- T2. Check that `SHORT_FLOAT'DIGITS < FLOAT'DIGITS` and `FLOAT'DIGITS < LONG_FLOAT'DIGITS` if these types are defined for an implementation.
- T3. Check that the values of `FIRST` and `LAST` can be assigned without raising `CONSTRAINT_ERROR`.
- T4. Check that the upper and lower bounds in a floating point type definition's range constraint can have different real types (including fixed point types).
- T5. For each of digits 5..29, check that values corresponding to `LARGE`, `-LARGE`, `SMALL`, `-SMALL`, `EPSILON`, and `1.0+EPSILON` can be assigned and used in equality relations.
Implementation Guideline: Check also that correct values are returned for generic formal types.
- T6. Check that negative powers of 2.0 down to $2.0^{(-30)}$ are represented exactly for digits 5..29.
- T7. For digits 5..29, check that a literal value that is not a model number lies in the correct model interval.
Implementation Guideline: Include a check for a generic formal type.
- T8. For digits 5..29, check that unnormalized literals representing model numbers are represented correctly.
- T9. Check that
- a. two types with the same textual declaration are distinct,
 - b. two types derived from a single type are distinct, and
 - c. subtypes of distinct types are distinct.
- T10. Check that the base type is determined partly by the range given in the type declaration.
Implementation Guideline: Use a type with digits 1 and whose lower and upper bounds are the safe numbers bounding the range for the most precise floating point type. Check that the declaration is accepted, and report whether the base type is different from a simple digits 1 base type.
- T11. Check that `CONSTRAINT_ERROR` is raised for a subtype indication having either of the forms, `T digits D` or `T digits D range L..R`, when `T'DIGITS < D`.
Check that `CONSTRAINT_ERROR` is raised for a subtype indication having the form `T digits D range L..R` if `L..R` is a non-null range and either `L` or `R` do not belong to `T's` range.
Check that no exception is raised otherwise.
Implementation Guideline: Perform the tests for generic formal types as well.
- T12. Check that explicit conversions, membership tests, and qualification for a subtype are evaluated with the accuracy of the base type.
Implementation Guideline: Include a check for generic formal types.
Check whether assignment for a subtype is performed with less precision than for the base type, and if so, whether range constraint checks are performed after the value to be assigned is given a suitable approximation for storage.
- T13. Check that the predefined floating point base types are equivalent to the predefined type, e.g., `FLOAT'FIRST = FLOAT(FLOAT'BASE'FIRST)`, and `FLOAT' DIGITS = FLOAT'BASE'DIGITS`.

3.5.8 Operations of Floating Point Types

Semantic Ramifications

S1. Since T'SAFE_LARGE gives the largest safe number for T'BASE, and since T'BASE'MANTISSA can be larger than T'MANTISSA, there is no attribute that reliably gives the largest safe number for T; the largest safe number for T can be less than T'SAFE_LARGE, since T's safe numbers can have fewer mantissa bits than the safe numbers for T'BASE. Similar reasoning applies to T'SAFE_SMALL, which can be smaller than the smallest positive safe number for T.

Changes from July 1982

S2. The attributes, SAFE_EMAX, SAFE_SMALL, and SAFE_LARGE, yield values for the base type of the prefix.

Changes from July 1980

S3. Attributes for safe numbers have been introduced.

S4. **abs** is an operator, not a function.

Legality Rules

- L1. The prefix of the attributes DIGITS, EPSILON, EMAX, and SAFE_EMAX must denote a floating point subtype (RM 3.5.8/3-13).
- L2. The prefix of the attributes MANTISSA, SMALL, LARGE, SAFE_SMALL, and SAFE_LARGE must denote either a fixed or floating point type (RM 3.5.8/3-13 and RM 3.5.10/5-12).

Test Objectives and Design Guidelines

- T1. Check that DIGITS, MANTISSA, EMAX, and SAFE_EMAX have type *universal_integer* and SMALL, LARGE, EPSILON, SAFE_SMALL and SAFE_LARGE have type *universal_real*.

Implementation Guideline: Include generic formal subtypes as a prefix.

Check that FIRST and LAST return values having the base type of their prefix.

Implementation Guideline: Include a null subtype as a prefix and a generic formal type.

For a static subtype prefix, check that the attributes can be used in static expressions.

- T2. Check the values of the attributes for all digits N, N=5..29 and check that the following relations hold:

- $T'DIGITS \leq T'BASE'DIGITS$
- $T'MANTISSA \leq T'BASE'MANTISSA$
- $T'EPSILON \geq T'BASE'EPSILON$
- $T'EMAX = 4 * T'MANTISSA \leq T'SAFE_EMAX$
- $T'SAFE_EMAX = T'BASE'SAFE_EMAX$
- $T'SAFE_SMALL = 2.0^{**}(-T'SAFE_EMAX - 1)$
- $T'SAFE_LARGE = 2.0^{**}T'SAFE_EMAX - 2.0^{**}(T'SAFE_EMAX - T'BASE'MANTISSA)$

Implementation Guideline: Compute the above formula as $2^{**}(T'SAFE_EMAX - 1) - 2^{**}(T'SAFE_EMAX - T'BASE'MANTISSA) + 2^{**}(T'SAFE_EMAX - 1)$, to avoid the possibility of overflow.

Implementation Guideline: Include a check for generic formal types.

- T3. Check that the prefix of DIGITS, EPSILON, EMAX, and SAFE_EMAX cannot be a fixed point type.

3.5.9 Fixed Point Types

Semantic Ramifications

- S1. The operations declared by a fixed point type definition are:

```

basic operations (RM 3.5.10/1)
  assignment
  membership tests
  qualification
  explicit conversion (from any numeric type)
  implicit conversion (from universal_real)
operators (RM 3.5.10/14)
  relational operators
  unary + and -
  binary + and -
  * for one operand having an integer type (RM 4.5.5/7)
  / for divisor having an integer type (RM 4.5.5/7)
  abs
attributes
  ADDRESS (RM 13.7.2/3)
  BASE (RM 3.3.3/9)
  FIRST (RM 3.5/8)
  LAST (RM 3.5/9)
  SIZE (RM 13.7.2/4)
  DELTA (RM 3.5.10/4)
  MANTISSA (RM 3.5.10/5)
  SMALL (RM 3.5.10/6)
  LARGE (RM 3.5.10/7)
  FORE (RM 3.5.10/8)
  AFT (RM 3.5.10/9)
  SAFE_SMALL (RM 3.5.10/11)
  SAFE_LARGE (RM 3.5.10/12)
  MACHINE_ROUNDS (RM 13.7.3/3)
  MACHINE_OVERFLOW (RM 13.7.3/4)

```

The operators for multiplying or dividing two fixed point values are declared in STANDARD (RM 4.5.5/9); these operators are not implicitly declared after a fixed point type declaration (see IG 3.5.10/S for further discussion).

S2. A fixed point type is implemented as a scaled binary number, where the binary point separating the integer and fractional part can be considered to lie either between the represented bits or outside the represented bits (either to the left or right). The position of the binary point is determined by the scale factor.

S3. An implementation is only required to support scale factors that are integral powers of two, since an implementation can choose to restrict its support for representation clauses to those that are easily implemented (RM 13.1/10). By allowing only integral powers of 2 as scale factors (in the representation clause for 'SMALL'), the implementation of fixed point multiplication and division is considerably simplified. On the other hand, applications that actually make use

of fixed point values will find such a restriction unhelpful, since hardware devices often produce or require fixed point values with scale factors that are other than powers of two.

S4. The fixed point type, DURATION (see RM 9.6/4), requires a mantissa of at least 24 bits, since it must cover the range $-86_400 .. 86_400$ with DURATION'SMALL being no larger than 0.020. This means fixed point operations such as multiplication and division must be supported for fixed point values having at least 24 bits of accuracy. A double precision integer representation must often be used in this case.

S5. The requirement that an implementation have at least one anonymous fixed point type (RM 3.5.9/7) means that there are at least two fixed point types whose scope includes any arithmetic expression (the type DURATION and at least one other, anonymous, type). Consequently, if V is a variable of type DURATION, $V * 60.0$ is illegal because there is no unique fixed point type to which the literal 60.0 can be converted (see further discussion in IG 4.5.5.b/S); such an expression must be written either as DURATION ($V * \text{DURATION}(60.0)$) or as $V * 60$ (multiplication of a fixed point value by an integer is allowed and yields the same fixed point type).

S6. SYSTEM.FINE_DELTA is defined as the smallest delta allowed in a fixed point constraint that has the range constraint $-1.0 .. 1.0$ (RM 13.7.1/6). This value, in essence, determines the largest fixed point mantissa value an implementation is prepared to support, e.g., if fixed point values can be represented with at most 31 bits of precision, SYSTEM.FINE_DELTA should be $2^{*(-31)}$.

S7. SYSTEM.FINE_DELTA is not the smallest delta that an implementation supports. For example, consider:

```
type T is delta SYSTEM.FINE_DELTA range -1.0 .. 1.0;
type U is delta SYSTEM.FINE_DELTA/2 range -0.5 .. 0.5;
```

This declaration should be considered legal because it can be represented using the same mantissa length as T. (In this case, the implicit binary point lies to the left of the most significant digit in the representation.) Similarly, the following declaration should be accepted:

```
type NT is delta 2*SYSTEM.FINE_DELTA range -2*T/LARGE .. 2*T/LARGE;
```

S8. Suppose fixed point types are represented as signed, twos-complement, 31-bit numbers. i.e., SYSTEM.MAX_MANTISSA is 31. Then the following type declaration cannot be rejected:

```
type T is delta 2.0**(-31) range -1.0 .. 1.0;
```

Although the upper bound, 1.0, cannot be represented physically (since all 31 bits are used to represent values lying between -1.0 and $1.0 - \text{T'SMALL}$) the requirement is that T's largest model number lie "at most" T'SMALL from the upper bound specified in T's declaration. If the value of the largest model number is $1.0 - \text{T'SMALL}$ (i.e., $1.0 - 2.0^{*(-31)}$), this condition is satisfied even though 1.0 is not representable. T'LAST will equal $1.0 - \text{T'SMALL}$, and the comparison $1.0 = \text{T'LAST}$ can raise an exception when 1.0 is converted to T's base type.

S9. Now consider the declaration:

```
type U is delta 2.0**(-30) range -1.0 .. 1.0;
```

The model numbers for U only have 30-bit mantissas. If the hardware fixed point type uses 31 bits, the implementation has the option of putting the extra bit on the left (increasing the range of representable values, and in particular, making 1.0 a representable number), or on the right (increasing the precision with which values are actually represented). Although the RM imposes no requirement on where extra bits are to be placed, it is preferable if the implementation provides extra precision rather than extra range; this is what most fixed point application programmers would expect. (Providing extra precision is allowed; see AI-00341.)

S10. The RM defines the default value of SMALL to be the largest power of two consistent with the specification given after delta:

```
type T is delta 0.01 range -100.00 .. 100.00;
X : T := T'SMALL;
```

Unless a representation clause is given for T'SMALL, the value of T'SMALL must be $2^{**}(-7) = 0.0078125$. With such a value of SMALL, 0.01 is not a model number, so $50 * X$ will not equal the model number 0.5. To make 0.01 a model number, a length clause must be given:

```
for T'SMALL use 0.01;
```

If this clause is accepted by an implementation (see RM 13.1/10), 0.01 will be a model number, and so will $50 * X$.

S11. Specifying a decimal value for 'SMALL allows an implementation to use a decimal representation for fixed point values and operations. However, care must be taken to ensure that the required range of model numbers is supported. For example:

```
type FIX is delta 0.1 range -99.5 .. 99.5;
for FIX'SMALL use 0.1;
```

Since the model numbers use a binary representation, the set of model numbers will be $-(1023 * 0.1) .. 1023 * 0.1$. Computations using the predefined arithmetic operations cannot overflow when computing $99.5 + 0.5$, for example, since the result lies in the range of model numbers for the type. To ensure that an exception is not raised incorrectly, an implementation will generally have to perform computations using at least four decimal digits. On the other hand, three decimal digits will suffice to hold stored values of type FIX since these values can never lie outside the range $-99.5 .. 99.5$.

S12. If type FIX is represented with binary numbers instead of fixed decimal, the set of model numbers is, of course, unchanged. The value $1023 * 0.1$ requires ten bits (plus sign), and the value $995 * 0.1$ also requires ten bits, so there is no difference in size between stored values of type FIX and values of the base type.

Changes from July 1982

S13. There are no significant changes.

Changes from July 1980

S14. Safe numbers are defined for fixed point types.

S15. The default value of SMALL is no longer determined partly by the implementation. In particular, it must be a power of 2 if there is no representation clause for SMALL.

Legality Rules

- L1. The arithmetic operators mod, rem, and ** are not predefined for fixed point types (RM 3.5.10/14).
- L2. The expression following delta must be a static expression having either a fixed point type, a floating point type, or the type *universal_real* (RM 3.5.9/3). The value of the expression must be greater than zero (RM 3.5.9/3).
- L3. If a fixed point constraint is used as a real type definition, a range constraint must be provided and the bounds of the range constraint must be given by static expressions having either a fixed point type, a floating point type, or the type *universal_real*; the expressions need not have the same real type (RM 3.5.9/3).

- L4. The number of bits required to represent the model numbers of the type must not exceed `SYSTEM.MAX_MANTISSA` (RM 13.7.1/5).
- L5. If a fixed point constraint follows a `t` mark in a subtype indication, the type mark must denote a fixed point subtype (RM 3.5.9/13), and the subtype indication must not appear in an allocator (RM 4.8/4).

Exception Conditions

- E1. If a fixed accuracy definition appears in a subtype indication whose type mark is `T`, `CONSTRAINT_ERROR` is raised if the value of the expression following `delta` is less than `TSMALL` (RM 3.5.9/13 and RM 3.3.2/9).
- E2. If a fixed accuracy definition with a range constraint appears in a subtype indication whose type mark is `T`, `CONSTRAINT_ERROR` is raised if the specified range is not null and one (or both) of the bounds does not lie in the range `TFIRST .. TLAST` (RM 3.5.9/13, RM 3.5/3, and RM 3.3.2/9).

Test Objectives and Design Guidelines

T1. Check that

- none of the expressions in a fixed accuracy constraint can have the type integer.
- the expression following `delta` must be static and have a value greater than zero.
- a range constraint must be present in a fixed point type definition.
- a fixed point type definition must be rejected if it requires more than `SYSTEM.MAX_MANTISSA` bits.

Implementation Guideline: For example, type `T` is `delta SYSTEM.FINE_DELTA range -2.0 .. 2.0`. A similar declaration can be constructed using `SYSTEM.MAX_MANTISSA`.

T2. Check that:

- the correct default value of *small* is used.
Implementation Guideline: Include cases where the specified `delta` is not a power of two.
- the expression following `delta` and the bounds in a fixed point type declaration can all have different real types, including floating point.
- a type with only 1 model number is handled correctly.
Implementation Guideline: For example, type `T` is `delta 1.0 range -1.0 .. 1.0`, only has the model number 0.0.
- the binary point in the mantissa can lie outside the mantissa (either to the left or to the right).

T3. Check whether the base type of a fixed point type provides extra precision or extra range.

T4. Check that `CONSTRAINT_ERROR` is raised for a subtype indication having the form `T delta S` or `T delta S range L .. R` if `TSMALL` is greater than `S`.

Check that `CONSTRAINT_ERROR` is raised for a fixed point subtype indication containing a non-null range constraint if either bound does not belong to the subtype being constrained.

Implementation Guideline: Repeat these tests for generic formal types.

3.5.10 Operations of Fixed Point Types

Semantic Ramifications

S1. For a fixed point subtype, the value of MANTISSA depends both on the value of the expression given for SMALL and the bounds of the range. Since, for a subtype declaration, the range need not be static, the result returned by T'MANTISSA must, in general, be computed at run-time. For example, consider:

```
type T is delta 0.25 range -4.0 .. 4.0;
subtype ST1 is T delta 0.25 range 1.0 .. 2.0;
subtype ST2 is T range 1.0 .. 2.0;
subtype SU is T delta 1.5 range -4.0 .. 4.0;
```

The model numbers for T, and hence T'MANTISSA, are determined by the value specified for delta and the range. In this case, T'SMALL is 0.25; T'MANTISSA is 4, with model numbers ranging from -3.75 .. 3.75. The model numbers for ST1 are determined by the reduced range since ST1'SMALL = T'SMALL = 0.25. For the reduced range, the model numbers need only range from -1.75 .. 1.75, so ST1'MANTISSA is 3. If either bound for ST1 were nonstatic, ST1'MANTISSA would have to be computed at run time. The model numbers for ST2 are the same as the model numbers for T, since there is no fixed accuracy constraint in ST2's declaration (RM 3.5.9/14). For SU, SU'SMALL is 1.0 (the smallest power of two less than 1.5). The model numbers for SU cover the range -3.0 .. 3.0, so SU'MANTISSA is 2.

S2. The model numbers for a subtype can only be redefined if a fixed accuracy definition is present. For example:

```
type F1 is delta 1.0 range -100.0 .. 100.0;
subtype F2 is F1 range -50.0 .. 50.0;
subtype F3 is F2 delta 2.0;                      -- new model numbers
subtype F4 is F1 delta 2.0;                      -- new model numbers
subtype F5 is F4 range -50.0 .. 50.0;
```

F1'MANTISSA is 7 and the model numbers cover the range $-(2^{**7}-1) .. 2^{**7} - 1$. The range of F2 is halved, but the model numbers for F2 are not affected; F2 only contains a range constraint, not a fixed point constraint. Therefore, F2'MANTISSA is still 7. F3'MANTISSA, however, is 5 because the value of *small* has been doubled and the range for F3 is half the range of F1. F4'MANTISSA is 6 because just the value of *small* has doubled. F5'MANTISSA is also 6 since F5 has the same model numbers as F4. Note, however, that although F3'DELTA = F5'DELTA and F3 and F5 have the same range, F3'MANTISSA = 5 and F5'MANTISSA = 6 — they have different model numbers even though the value of delta and the range are the same.

S3. The value of the attribute 'FORE is calculated in terms of the decimal representation of model numbers belonging to the subtype. Consider the following fixed point type declaration:

```
type F is delta 0.1 range 0.0 .. 9.96;
for F'SMALL use 0.01;
```

The model numbers belonging to F cover the range 0.0 .. 9.96. The value of F'FORE is 2. In this case, the value of F'FORE is unsuitable for output with an AFT of 1, since the decimal points will not be aligned — allowed for the integer part.

S4. The value returned by FORE can be implemented as follows:

```
type G is delta 0.01 range
for G'SMALL use 0.01
```


AD-A189 647

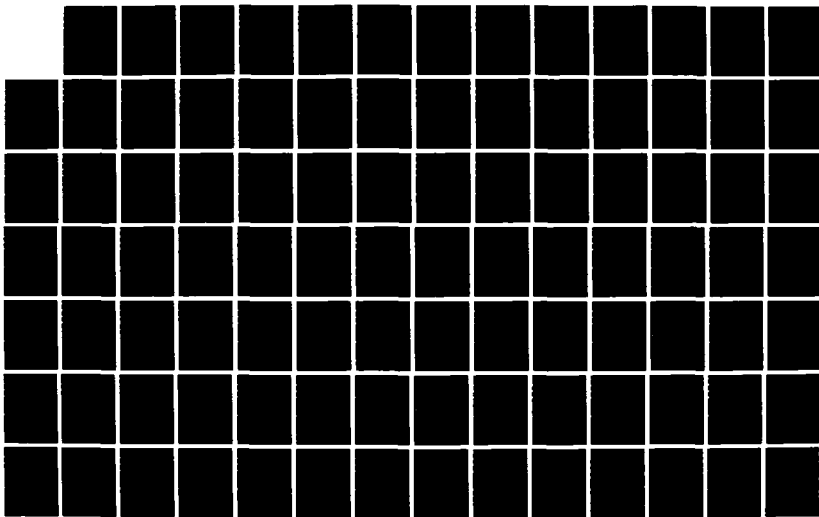
THE ADA (TRADE NAME) COMPILER VALIDATION CAPABILITY
IMPLEMENTERS' GUIDE VERSION 1(U) SOFTECH INC WALTHAM MA
J B GOODENOUGH DEC 86

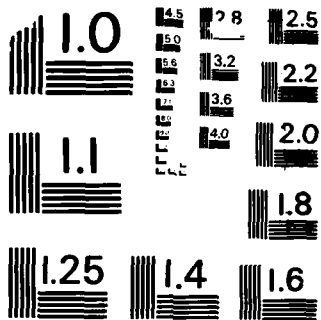
2/9

UNCLASSIFIED

F/G 12/5

NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

subtype SG is G delta 0.1 range 1.0 .. 9.995;

For subtype SG, 9.99 and 10.00 are consecutive model numbers (RM 3.5.9/16). It is implementation-dependent whether the upper bound of SG is represented as the model number 9.99 or as the model number 10.00. Depending on the implementation's choice, the value returned by SG'FORE will be either 2 or 3. Of course, the bounds of SG need not be given by static expressions. If the upper bound is nonstatic and has a value lying in the model interval 9.99 to 10.00, SG'FORE's value will be implementation dependent (and must be computed at run time).

s5. The operations for multiplying or dividing two fixed point values are declared in STANDARD, not after a particular fixed point type declaration (RM 4.5.5/9). The other multiplication operators (integer * fixed and fixed * integer) and the other division operator (fixed/integer) are declared immediately after the fixed point type declaration. The place where these operators are declared affects their visibility and how expanded names can be used. For example:

```
package P is
  type T is delta 0.1 range -100.0 .. 100.0;
end P;

X1 : P.T := 1.0;
X2 : P.T := 2.0;
X3 : P.T := P.T(X1 * X2);      -- legal
X4 : P.T := P.T(P.""(X1, X2)); -- illegal
X5 : P.T := 2 * X1;            -- illegal
```

The use of "" in the initialization of X5 is illegal because this operator is declared inside package P and is not directly visible, since no use clause for P has been written. On the other hand, the use of "" in the initialization of X3 is legal since this operator is declared in STANDARD, and so is directly visible; correspondingly, the use of P."" in X4's initialization is illegal since no "" operator is declared in P that has operands of type P.T.

Changes from July 1982

- s6. The value of FORE is defined to be at least 2.
- s7. An explicit definition of AFT is given.
- s8. The SAFE attributes are defined to return values of the base type.

Changes from July 1980

- s9. The attribute ACTUAL_DELTA is now called SMALL.
- s10. The attribute BITS is now called MANTISSA.
- s11. The attributes FORE and AFT have been introduced.
- s12. Attributes for the safe numbers have been introduced.

Legality Rules

- L1. The prefix of the attributes DELTA, FORE, and AFT must denote a fixed point subtype (RM 3.5.10/3-12).
- L2. The prefix of the attributes MANTISSA, SMALL, LARGE, SAFE_SMALL, and SAFE_LARGE must denote either fixed or floating point types or subtypes (RM 3.5.8/3-13 and RM 3.5.10/5-12).

Test Objectives and Design Guidelines

- T1. Check that the prefix of DELTA, FORE, and AFT cannot be a floating point type.
- T2. Check that T'DELTA yields correct values for subtype T.
Implementation Guideline: Include a case where T has been declared by a subtype declaration.
Implementation Guideline: Check for generic formal types also.
- T3. Check that T'MANTISSA has the correct value.
Implementation Guideline: Include cases where T'MANTISSA must be computed at run-time.
Implementation Guideline: Check for generic formal types also.
- T4. Check that correct values are yielded for T'SMALL and T'LARGE.
Implementation Guideline: Check for generic formal types also.
- T5. Check that T'FORE and T'AFT yield correct values.
Implementation Guideline: Include cases where the value of FORE must be computed at run-time.
Implementation Guideline: Include cases where T'DELTA is greater than 0.1 and less than 0.1.
Implementation Guideline: Include cases where FORE is likely to yield an inconvenient value.
- T6. Check that SAFE_SMALL and SAFE_LARGE return appropriate values.
Implementation Guideline: Check for generic formal types also.
- T7. Check that FIRST and LAST return correct values.
Implementation Guideline: Include cases where the prefix denotes a null subtype and a generic formal type.
- T8. Check that the multiplication and division operators for two fixed point operands are declared in STANDARD and can be directly visible even when other fixed point operators are not directly visible.

3.6 Array Types

Semantic Ramifications

S1. The syntax requires that the indices of an array type be either all constrained or all unconstrained. For example, the following is illegal:

```
type MATRIX is array (INTEGER range <>, BOOLEAN) of FLOAT;  -- illegal
```

S2. The index subtype of all the following array types is INTEGER range 1..3:

```
subtype R1_3 is INTEGER range 1..3;
```

```
type A1 is array (1..3) of INTEGER;
```

```
type A2 is array (INTEGER range 1..3) of INTEGER;
```

```
type A3 is array (R1_3 range <>) of INTEGER;
```

```
X1 : A1;
```

```
X2 : A2;
```

```
X3 : A3 (1..3);
```

Because the index subtype is 1..3, only choice values in this range are allowed in non-null aggregates, i.e., the aggregates in each of the following assignment statements will raise CONSTRAINT_ERROR (RM 4.3.2/11):

```
X1 := (2..4 => 0);  -- CONSTRAINT_ERROR
X2 := (2..4 => 0);  -- CONSTRAINT_ERROR
X3 := (2..4 => 0);  -- CONSTRAINT_ERROR
```

S3. The prefix, *component*, in the syntactic category name *component_subtype_indication*, is only present so that the rules concerning the array component's subtype can be phrased in a more readable manner. No particular semantic restriction is associated with the prefix. (Compare the use of this prefix with the use of *discrete* in *discrete_subtype_indication* (in the syntax rule for discrete range). The prefix in this case does reflect the restriction that the subtype indication specify a discrete type (RM 3.6.1/2).)

S4. If the number of components in a dimension exceeds SYSTEM.MAX_INT, the elaboration of a type or object declaration array can raise NUMERIC_ERROR (or CONSTRAINT_ERROR; see AI-00387):

```
type MAX is range -1 .. SYSTEM.MAX_INT;
type ARR is (MAX range <>) of BOOLEAN;      -- no exception here

subtype A_MAX is ARR(MAX);      -- can raise NUMERIC_ERROR
X : ARR (MAX);                  -- can raise NUMERIC_ERROR
```

NUMERIC_ERROR can be raised because some implementations will store the length of a dimension as part of an array's representation. This length might be computed when the array subtype declaration is elaborated or when each object declaration is elaborated. If this length exceeds SYSTEM.MAX_INT, then its value cannot be represented accurately, and NUMERIC_ERROR can be raised (RM 11.1/6). (Of course, the object declaration might raise STORAGE_ERROR if NUMERIC_ERROR is not raised.)

S5. A multidimensional array type is not semantically equivalent or similar to an array of array type. In particular, the rules concerning multidimensional array aggregates are different from those for array of array aggregates (see IG 4.3.2/S).

Approved Interpretations

Array index types must be discrete (AI-00249).

Changes from July 1982

S6. There are no significant changes.

Changes from July 1980

S7. The concept of "index subtype" is defined and used to clarify the rules concerning allowable values of choices in aggregates.

Legality Rules

L1. Arrays must be declared with discrete index types (AI-00249).

Exception Conditions

NUMERIC_ERROR (or CONSTRAINT_ERROR; see AI-00387) can be raised if the number of components for a dimension exceeds SYSTEM.MAX_INT (RM 11.1/6).

Test Objectives and Design Guidelines

- T1. Check that an array type cannot be declared with a nondiscrete index type.
Implementation Guideline: Declare both constrained and unconstrained array types.
- T2. Check that the indices of multidimensional array types must be either all constrained or all unconstrained.
- T3. Check whether an array type or subtype declaration raises NUMERIC_ERROR (or CONSTRAINT_ERROR) when one dimension of the type has more than SYSTEM.MAX_INT components.

Implementation Guideline: Check for one- and two-dimensional arrays. For two-dimensional arrays, the oversize dimension should appear once as the first dimension and once as the second.

- T4. Check that the index subtype of an array is correctly determined for constrained and unconstrained array types.

Implementation Guideline: Use all forms of discrete range in the constrained array cases.

3.6.1 Index Constraints and Discrete Ranges

This section discusses discrete ranges and index constraints separately. Since index constraints use discrete ranges, discrete ranges are discussed first.

3.6.1.a Discrete Ranges

Semantic Ramifications

- S1. A discrete range can have any of the following forms:

- ST, where ST is a discrete type mark.
- ST range L .. R
- L .. R, where L and R are expressions having discrete types.
- A'RANGE(N), where A is an array object, array value, constrained array type, or access type designating a constrained array type.

- S2. A discrete range can be used to specify:

- a choice in a case statement, variant part, or aggregate;
- a slice;
- a loop parameter;
- an index constraint in a subtype indication or constrained array type definition;
- a family of entries.

In loop parameter specifications, constrained array definitions, and declarations of entry families, a discrete range defines the type of the loop parameter, array, or entry family index, respectively, as well as constraining the set of acceptable values. In the other contexts, a compiler must determine whether the discrete range has, or can be given, a type consistent with the type required by the context. The form A'RANGE(N) cannot be used in a context where a static choice is required, namely, in the choice of a case statement or variant part or in an array aggregate with more than one component association.

- S3. We will first discuss discrete ranges (other than the form A'RANGE) in which a type mark appears explicitly, and then the form in which just the range is given.

Discrete Ranges with Type Marks

- S4. The type associated with a discrete range determines what values can be specified for a range. If the type mark in a discrete range is a subtype name, ST, ST's base type is the type associated with the discrete range. For example, given

```
type DAY is (MON, TUE, WED, THU, FRI, SAT, SUN);
subtype WEEKDAY is DAY range MON .. FRI;
subtype MIDWEEK is WEEKDAY range TUE .. THU;
```

the discrete range

MIDWEEK range WED .. THU

has the type DAY. RM 3.5/4 states that *only* values of MIDWEEK's base type, DAY, can appear in this discrete range. In addition, for a discrete range of the form:

ST range L .. R

if R is greater than or equal to L (i.e., the range is not null), CONSTRAINT_ERROR is raised if the values of L and R do not satisfy the range constraints of ST (RM 3.5/4 and RM 3.3.2/9). For example:

MIDWEEK range WED .. FRI

Evaluation of this discrete range raises the exception CONSTRAINT_ERROR, since FRI is not in MIDWEEK's set of values.

Discrete Ranges Without Type Marks

S5. When a discrete range has the form L .. R and each bound has type *universal_integer* and is either a numeric literal, named number, or attribute, and the discrete range is used:

- in a loop parameter specification, or
- as an index in a constrained array definition, or
- in the declaration of a family of entries,

L and R are implicitly converted to the predefined type INTEGER. For example, in a loop statement:

for I in 1 .. 10 loop

I cannot have the type SHORT_INTEGER, even if an implementation supports this type and SHORT_INTEGER'LAST > 10. Moreover, if 1_000_000 is greater than INTEGER'LAST, then 1 .. 1_000_000 raises NUMERIC_ERROR (or CONSTRAINT_ERROR; see AI-00387) in these contexts, since the discrete range is equivalent to

for I in INTEGER range 1 .. 1_000_000 loop

and the implicit conversion of 1_000_000 to INTEGER raises NUMERIC_ERROR (RM 4.6/15) (or CONSTRAINT_ERROR; AI-00387).

S6. In the constructs mentioned above, if L and R are not each either numeric literals, named numbers, or attributes, they must both have the same discrete type *other than universal_integer*. This rule makes the discrete range -1 .. 1 illegal because -1 is an expression containing a unary operator; it is not a numeric literal, and so the rule specifying implicit conversion to INTEGER does not apply. The second rule (in RM 3.6.1/2) says the type must be determined using the requirement that the bounds of a discrete range must have the same discrete type; no other information derived from the context in which the discrete range appears can be used (see RM 8.7/7-13 and IG 8.7.a.7/S).

S7. In other constructs, additional rules can (and must) be used to resolve any uncertainty in the type of L and R:

- in case statements, the type of each choice must match the type of the case expression (which cannot be overloaded; RM 5.4/3);
- in variant parts, the type of a choice is determined by the type of the discriminant (the discriminant cannot be ambiguous in its type);

- in an array aggregate, all the choices must have the type of the index and the type of the aggregate is determined by the context in which it appears, not by the form or type of its choices (RM 4.3/7);
- in a slice, the type of the discrete range is determined by the type of the array being sliced and the type of the bounds of the slice; (When the result of a function is sliced, the type of the array returned by the function is contextually determined and may depend on the type of the discrete range used in the slice; see IG 8.7.b/S23.)
- in an index constraint, the type of the discrete range is determined by the type of the array being constrained and the type of the bounds for each dimension.

The above rules must be sufficient to uniquely determine the type of L and R; otherwise the program is illegal.

Overloading Resolution

S8. Both the lower and upper bound of a range must have the same type. This rule must be taken into account when resolving the type of overloaded literals or expressions used for bounds. For example, if the enumeration type

```
type SOL is (SUN, MERCURY, VENUS, EARTH, MARS, JUPITER,
             SATURN, URANUS, NEPTUNE, PLUTO);
```

is in the same scope as DAY, then SUN .. TUE is a discrete range of type DAY and SUN .. VENUS is a discrete range of type SOL.

S9. The rule that the discrete range must be a discrete type can be used to resolve an overloading ambiguity. For example, if F and G are parameterless functions that return either INTEGER or FLOAT values, then

```
for I in F .. G loop ... end loop;
```

is unambiguous because only functions returning discrete values are legal here. If we introduce a new type and function definition:

```
type T is new INTEGER;
function F return T;
```

then

```
for I in 2 .. F loop
```

is ambiguous, since 2 and F can both either be of type T or type INTEGER.

S10. Overloading resolution is discussed further in IG 8.7/S.

Changes from July 1982

S11. If a discrete range has the form L .. R and both L and R have the type *universal_integer*, then L and R must be either a numeric literal, a named number, or an attribute when the discrete range is used in a constrained array type definition, a loop parameter specification, or the declaration of a family of entries.

Changes from July 1980

S12. The order of evaluation for the bounds of a discrete range is explicitly not defined by the language.

Legality Rules

- L1. For a discrete range having the form of a type mark, the type mark must denote a discrete type (RM 3.6.1/2).
- L2. For a discrete range having the form ST range L .. R, ST must denote a discrete type (RM 3.6.1/2) and L and R must have ST's base type (RM 3.5/4).
- L3. For a discrete range having the form L .. R, L and R must have the same discrete type (RM 3.6.1/2).
- L4. The type of the bounds of a discrete range having the form L .. R must be determined independently of the context when the discrete range is used in a constrained array definition, a loop parameter specification, or the declaration of a family of entries (RM 3.6.1/2).
- L5. If a discrete range has the form L .. R and both L and R have the type *universal_integer*, then L and R must be either a numeric literal, a named number, or an attribute when the discrete range is used in a constrained array type definition, a loop parameter specification, or the declaration of a family of entries (RM 3.6.1/2).
- L6. For a discrete range having the form A'RANGE(N), A must be an array object, an array value, a constrained array type, or an access type that designates a constrained array type. N must be a static expression having the type *universal_integer* and must have a value greater than zero and not exceeding the number of dimensions of the array (RM 3.6.2/2, RM 3.6.2/8, and RM 3.8.2/2).

Exception Conditions

- E1. For a discrete non-null range of the form ST range L .. R, CONSTRAINT_ERROR is raised if L or R is outside the range of ST but within the range of ST's base type (RM 3.5/4 and RM 3.3.2/9).
- E2. For a discrete range of the form L .. R where L and R are integer literals, named numbers having type *universal_integer*, or attributes returning a value of type *universal_integer*, NUMERIC_ERROR (or CONSTRAINT_ERROR; see AI-00387) is raised if either L or R lies outside the range of INTEGER and the discrete range is used in a constrained array type definition, a loop parameter specification, or the declaration of a family of entries (RM 4.6/15 and RM 3.6.1/2).

Test Objectives and Design Guidelines

The contextually determined validity of discrete ranges is checked in each context in which discrete ranges can be used. We only check here for context-independent discrete range validity.

- T1. Check that the type mark (if present) of a discrete range and both bounds must be discrete types.

Implementation Guideline: Check all illegal combinations of the following:

- type mark non-discrete/discrete/absent
- lower bound non-discrete/discrete
- upper bound non-discrete/discrete
- bounds are literals/non-literals

Implementation Guideline: All combinations should be tried for discrete ranges in both loops and array type definitions. At least two illegal combinations should be tried for discrete ranges in case statement choices, variant part choices, array aggregates, slices, and as index constraints in an object declaration and a type declaration. Different forms of illegal discrete ranges should be used in each of these checks.

- T2. Check that the upper and lower bounds of a discrete range cannot have different discrete types. Use forms with and without a type mark.

Implementation Guideline: Try at least one example of a discrete range in each of the contexts permitting discrete ranges, namely, subtype indications, membership tests, aggregates, case statement choices, variant part choices, array or entry family declarations, slices, and loop parameter specifications.

- T3. Check that when a type mark is present in a discrete range and both bounds have the same type, the type of the bounds must be the type of the type mark.

Implementation Guideline: Use both a subtype and type name for the type mark, and use non-null ranges; limit this test to loops and array type definitions.

- T4. Check that for the form ST range L .. R, CONSTRAINT_ERROR is raised if the range is non-null and L or R are outside ST'FIRST .. ST'LAST but within the range of ST'BASE.

Check that no exception is raised if $L > R$ and L and R both belong to ST'BASE.

Implementation Guideline: Use both static and nonstatic out-of-range expressions for L and R, but write separate tests for the static and nonstatic cases. For null ranges, use both enumeration and INTEGER subtypes in loops, slices, and membership operations. For non-null ranges, check all contexts in which a nonstatic discrete range can appear, i.e., loops, array type definitions, aggregates with a single choice, slices, index constraints, and membership tests.

- T5. Check that when either L or R are not numeric literals, named numbers, or attributes, a discrete range of the form L .. R is illegal when used in a constrained array type definition, a loop parameter specification, or the declaration of a family of entries.

Using non-null discrete ranges of the form L .. R in loop, entry families, and array type definitions, check that NUMERIC_ERROR or CONSTRAINT_ERROR is raised if L and R are numeric literals, named numbers, or attributes and at least one of them is not in the range of INTEGER values.

In a separate test, check that neither a loop parameter nor an array index is assumed to have the type SHORT_INTEGER if L and R are both in the range of SHORT_INTEGER values.

Implementation Guideline: Use the loop parameter as a subscript for an array having a SHORT_INTEGER index type.

- T50. Check that functions overloaded to return a discrete and nondiscrete result are considered unambiguous if used as discrete range bounds in a loop, entry family, or array type definition where the type of the discrete range is not given explicitly (see IG 8.7.b/T13).

- T51. Check that functions overloaded to return an INTEGER type and some other discrete type (including a derived integer type) are considered ambiguous if used as discrete range bounds in a loop, entry family, or array type definition where the type of the discrete range is not given explicitly (see IG 8.7.b/T13).

3.6.1.b Index Constraints

Semantic Ramifications

S1. An index constraint raises CONSTRAINT_ERROR if one of its discrete ranges is incompatible with the corresponding index subtype (RM 3.6.1/4 and RM 3.3.2/9). Compatibility of a range with a subtype is implicitly defined by RM 3.5/4, namely, if the discrete range is not null, both bounds must belong to the index subtype. If the range is null, neither bound need belong to the index subtype, e.g.:

```
X : STRING (-1..-10);           -- legal; no exception
Y : STRING (INTEGER'LAST..INTEGER'FIRST); -- legal; no exception
```

Changes from July 1982

S2. There are no significant changes.

Changes from July 1980

S3. Arrays with nonstatic bounds are now allowed as record components even if the bounds do not depend on a discriminant.

S4. The lower bound of a null range in an index constraint no longer must satisfy the range constraint imposed by the subtype of the index.

S5. The upper bound of a null range used in an index constraint need not be the predecessor of the lower bound.

Legality Rules

- L1. An index constraint can only be applied to a type mark denoting an unconstrained array type or an access type designating an unconstrained array type (RM 3.6.1/3).
- L2. One discrete range must be provided in an index constraint for every index of the array type being constrained (RM 3.6.1/3).
- L3. The base type of each discrete range in an index constraint must be the same as the base type of the array index for which the discrete range applies (RM 3.6.1/3).
- L4. An index constraint must be given in a subtype indication if the type mark denotes an unconstrained array type and the subtype indication is used in an object declaration (non-constant; RM 3.6.1/7) or in a component declaration of a record or array type definition (RM 3.6.1/6).
- L5. An index constraint must be given in an allocator containing an unconstrained array type mark if no initial value is specified in the allocator (RM 3.6.1/8).

Exception Conditions

- E1. CONSTRAINT_ERROR is raised if a non-null discrete range in an index constraint specifies a range of values L through R, and L or R is not in the index subtype's range (RM 3.6.1/4, RM 3.5/4, and RM 3.3.2/9) but does belong to the range of the index base type.

Test Objectives and Design Guidelines**T71. Check that**

- an index constraint cannot be applied to a scalar type, record type, private type implemented as an array type, or an access type designating any of these types.
- the number of discrete ranges in an index constraint cannot be less than or greater than the number of indices in the array type being constrained. Check for both array and access types.
- the base types of the discrete ranges cannot be different from the base types of the corresponding indices.
Implementation Guideline: Include a check using A'RANGE, including when A is a formal generic type, and when the corresponding index type is a formal generic type.
- an index constraint cannot be given in a subtype indication whose type mark denotes a constrained array type or access type designating a constrained array type.

Implementation Guideline: There are two ways to declare access types designating constrained array types:

```

type STC is access STRING (1..3);    -- 1
type ACC_STR is access STRING;
subtype CST is ACC_STR (1..3);       -- 2

```

- an index constraint cannot be given in an allocator for an access type designating a constrained array type (see IG 4.8/T2).
- an index constraint cannot be given in an actual generic parameter (see IG 12.3/T5).
- an index constraint cannot be omitted in a subtype indication whose type mark denotes an unconstrained array type if the subtype indication is used in a nonconstant object declaration, in declaring the component type in an array type definition (used in a nonconstant object declaration, type declaration, or generic formal parameter declaration), or in the declaration of a record component.
- an index constraint cannot be omitted in an allocator naming an unconstrained array type if no initial value is specified for the array being allocated (see IG 4.8/T1).
- an index constraint must contain at least one discrete range.
- an index constraint cannot contain a box symbol.

T72. Check that CONSTRAINT_ERROR is raised if one of the bounds of a non-null discrete range does not lie in the range of the index subtype.

Implementation Guideline: Include a case where an index defines a null multidimensional array and one range is non-null.

Check that no exception is raised for a null index constraint if the lower bound of a null discrete range does not lie in the range of the index subtype, but does lie in the range of the index base type.

Check that no exception is raised if the upper bound of a null range is not the predecessor of the lower bound, but belongs to the index base type.

Implementation Guideline: Check the above for discrete ranges having the permitted forms: ST range L .. R, L .. R, and A'RANGE(N).

Implementation Guideline: Check that no exception is raised for a null array so that ARR'LAST - ARR'FIRST < INTEGER'FIRST, e.g., STRING (INTEGER'LAST .. -1).

Implementation Guideline: Include a check for access types and generic formal types.

T73. Check that if an index constraint is not imposed when declaring a formal parameter of a subprogram or a generic unit, the bounds of the formal parameter are defined when the subprogram is invoked or the generic unit is instantiated (see IG 6.4.1/T6 and IG 12.3.2/T24).

T74. Check that an index constraint can be omitted when declaring a constant object with a subtype indication, and that the bounds of the object are determined by the bounds of the initial value.

Implementation Guideline: Use catenation of strings as initial values.

T75. Check that an index constraint can be specified in an allocator naming an unconstrained array type (see IG 4.8/T5).

T76. Check that an index constraint in a record component declaration can use the name of a discriminant of the record (see IG 3.7.2/T13 and IG 3.7.2/T17).

T77. Check that the bounds of an array object designated by an access type are defined by the allocator that creates the object (see IG 4.8/T5, /T6).

T78. Check that the bounds of a generic formal array parameter of mode in are defined by the generic actual parameter if the subtype is unconstrained (see IG 12.3.1/T21).

Check that the bounds of a generic formal array parameter are defined by the subtype when the subtype is constrained (see IG 12.3.1/T26).

T79. Check that in a renaming declaration, the bounds of the new name are those of the renamed object (see IG 8.5/T6).

T80. Check that an index constraint can have the form A'RANGE.

3.6.2 Operations of Array Types

Semantic Ramifications

S1. Array attributes are defined for objects and values (i.e., function calls) as well as for types (unlike the scalar attributes 'FIRST and 'LAST, which are only defined for types). In addition, RM 3.8.2/2 defines these attributes for access values that designate arrays. In essence, the prefix for one of these array attributes can be either an object or a value having an array type, or an object or a value having an access type whose designated type is an array type.

S2. Since a function call can be used as a prefix of an array attribute,

```
"&" ("ABC", "DE") 'FIRST
```

is legal but

```
("ABC" & "DE") 'FIRST
```

is not; ("ABC" & "DE") is not, syntactically, a function_call. It is an expression.

S3. Similarly,

```
"DE" 'FIRST
```

is illegal, since "DE" is not the name of an operator symbol (RM 6.1/3). Moreover, no operator symbol can be used as a prefix of an array attribute since no operator symbol can be declared with zero parameters or with default parameters (RM 6.7/2).

S4. Since INTEGER'IMAGE(N) is parsed as a function call (see IG 4.1.4/S), it can be given as the prefix of an attribute:

```
INTEGER'IMAGE(N) 'LENGTH
```

However, since STRING("ABC") is a qualified expression, not a function call or a name, it is not allowed as the prefix of an attribute.

S5. 'FIRST and 'LAST produce results whose type is that of the index, whereas 'LENGTH produces a result of type *universal_integer*. A'LAST - A'FIRST + 1 is a legal Ada expression only if A's index type is an integer type. However, the values of A'FIRST and A'LAST can always be converted to integer values by using the 'POS attribute. INDEX_TYPE'POS (A'LAST) - INDEX_TYPE'POS (A'FIRST) + 1 gives the length of a dimension, except that if A is a null array, A'LAST need not be the predecessor of A'FIRST, so a negative value could be produced by this expression. If A'LENGTH is zero, A'LENGTH should not raise NUMERIC_ERROR (or CONSTRAINT_ERROR; see AI-00387) even if A'LAST - A'FIRST would do so.

S6. If the number of components for a particular dimension is greater than SYSTEM.MAX_INT, then an implementation is allowed to raise NUMERIC_ERROR (or CONSTRAINT_ERROR; AI-00387) when A'LENGTH is evaluated (RM 4.10/5). Note that A'LENGTH is, in general, a

nonstatic *universal_integer* expression, and must potentially be computed using the longest available integer type (RM 4.10/5).

S7. In determining the bounds associated with A'RANGE, A is only evaluated once (RM 4.1/10), i.e., A'RANGE is not completely equivalent to A'FIRST .. A'LAST when A is a function call with side effects.

S8. A'RANGE is a range, not a subtype, and so A'RANGE'FIRST is not allowed. However, A'RANGE can be used in object declarations, e.g.:

```

type A is array (1 .. 10) of T;
X1 : A'RANGE;                -- illegal; A'RANGE not a type mark
X2 : INTEGER range A'RANGE;   -- legal; A'RANGE is a range

```

Since A'RANGE is syntactically a range (RM 3.5/2), the declaration of X2 is legal.

S9. Since 'LENGTH returns type *universal_integer*, an expression of the form A'LENGTH = 6 is not ambiguous. Equality is defined for *universal_integer* types (RM 4.10/2) and so an implicit conversion to some other integer type is not allowed (see IG 4.6/S and RM 4.6/15).

S10. Array attributes can be applied to formal parameters, and in particular, to unconstrained formal array parameters, since parameters are objects (RM 3.2/3):

```

type T is array (INTEGER range <>) of INTEGER;
A : T (1 .. 50);
procedure P (X : T);

```

Within P, one can write X'FIRST, X'LAST, and X'RANGE, etc. For the call P(A), these would yield the INTEGER values 1, 50, and the range INTEGER range 1..50. For a slice, P (A(5..10)), the results would be 5, 10, and INTEGER range 5..10, and similarly for a null slice.

S11. Since catenation and the relational operators are predefined for all one-dimensional arrays, the declaration of a one-dimensional array type creates implicit declarations of these operators in the declarative region containing the array type declaration. Hence, if a one-dimensional array type was declared in package P, the implicitly declared operators can be referenced as P."&" and P."=", etc. Furthermore, since declaration of a constrained array type in an object declaration creates an anonymous array type definition (RM 3.6/6-8), the following use of "&" is legal:

```

X : array (1 .. 2) of CHARACTER := 'A' & 'B';

```

The declaration is equivalent to:

```

subtype %1_2 is INTEGER range 1..2;
type %Base is array (%1_2 range <>) of CHARACTER;
-- "&" and other operations declared here
X : %Base (1..2) := 'A' & 'B';

```

The symbols %1_2 and %Base represent the implicit declarations defined by RM 3.6/6-8.

S12. 'SIZE is defined even for unconstrained array types and should return the size needed to store the largest array of the type (RM 13.7.2/5).

S13. Syntactically, A'RANGE(N) parses as a function_call whose function name is A'RANGE and whose argument is (N) (see RM 4.1.4/2 to see why A'RANGE(N) is not syntactically an attribute, and RM 6.4.2/2 and 4.1/2 to see how it is parsed as a function call).

Changes from July 1982

S14. An array value (i.e., a function call) can be a prefix of an array attribute.

Changes from July 1980

S15. The array attributes can now be used with prefixes having an access type whose designated type is an array type.

S16. The type of the argument to 'FIRST, 'LAST, 'LENGTH, and 'RANGE is *universal_integer*. The value of the argument must be greater than zero, but not greater than the dimensionality of the array.

S17. 'LENGTH yields a value of type *universal_integer*.

S18. Catenation is no longer defined for arrays of a limited type.

S19. The 'RANGE attribute no longer defines a scalar subtype.

Legality Rules

- L1. The assignment, aggregate formation, equality, and inequality operations are not defined for limited array types (RM 3.6.2/1, 12).
- L2. The slice operation is not defined for multidimensional array types (RM 3.6.2/1).
- L3. The string literal formation operation is only defined for one-dimensional arrays whose component type is a character type (i.e., an enumeration type containing at least one character literal; RM 3.5.2/1) (RM 3.6.2/1).
- L4. The catenation operation is only defined for one-dimensional unlimited array types (RM 3.6.2/12).
- L5. For arrays, ordering operations are only defined for one-dimensional arrays whose component type is an enumeration type or an integer type (RM 3.6.2/12).
- L6. For arrays, the unary operator, not, and the logical operations are only defined for one-dimensional arrays whose component type is the predefined type BOOLEAN or a type derived (directly or indirectly) from this predefined type (RM 3.6.2/12 and RM 3.5.3/1).
- L7. An attribute of the form 'FIRST(N), 'LAST(N), 'LENGTH(N), 'RANGE(N), 'LENGTH, or 'RANGE can only be applied to a constrained array type, an array value, or an object of an array type (RM 3.6.2/2), or to an access value designating an object of an array type (RM 3.8.2/2).
- L8. An attribute of the form 'FIRST or 'LAST can only be applied to an array type, an array value, or an object of an array type (RM 3.6.2/2) or to an access value designating an object of an array type (RM 3.8.2/2), or to a scalar type (RM 3.5/7-9).
- L9. The attributes 'FIRST, 'LAST, 'LENGTH, and 'RANGE can have at most one argument, and that argument must be a static expression of type *universal_integer* having a value greater than zero and less than or equal to the number of dimensions defined for the array type (RM 3.6.2/2).

Exception Conditions

- E1. CONSTRAINT_ERROR is raised if the prefix of any array attribute has the access value, null (RM 4.1/10).

Test Objectives and Design Guidelines

- T1. Check that

- array attributes cannot be applied to an unconstrained array type;

Implementation Guideline: Include use with the attribute 'BASE to produce an unconstrained array type.

- the parameterized attribute forms cannot be applied to a scalar type or a scalar object;
- the attributes cannot be applied to a record or private type;
- more than one argument is not permitted;
- the attributes' argument cannot be a nonstatic expression;
- the value of the attributes' argument cannot be zero, negative, or greater than the number of dimensions of the relevant array type;
- the type of the attributes' argument must be *universal_integer*.

- T2. Check that `NUMERIC_ERROR` (or `CONSTRAINT_ERROR`; see AI-00387) may be raised if `A'LENGTH` is used in a context that requires an `INTEGER` value, and the number of components for a dimension exceeds `INTEGER'LAST`.

Implementation Guideline: Declare an array type A, for example, with bounds `-1..INTEGER'LAST` and check `A'LENGTH` in a context such as:

```
X := A'LENGTH - 2;
```

where X is of type `INTEGER`. Use an array of null records. Note that `NUMERIC_ERROR` need not be raised if the correct result can be computed and stored in X (RM 3.5.4/10).

Check that `'LENGTH` does not raise `NUMERIC_ERROR` (or any other exception) when applied to a null array A, even if `A'LAST - A'FIRST` would raise `NUMERIC_ERROR`.

Implementation Guideline: Declare a null array type using the longest integer type. Remember that the type declaration may raise `NUMERIC_ERROR` or `CONSTRAINT_ERROR` (AI-00387).

- T3. Check that `'LENGTH` yields a result of type *universal_integer*.

- T4. Check that each of the parameterized and unparameterized array attributes yield the correct values or range when applied to an array type, an array value, an array object, or an access value designating an array object.

Implementation Guideline: Include checks of generic formal array parameters and array types whose indexes are given by formal scalar types.

Check that for `A'RANGE`, the prefix A is only evaluated once.

Check that the result of `"&"` can be used as the prefix to an array attribute when `"&"` is written explicitly as a function call.

Check that the `'RANGE` attribute can be used as the range in a subtype indication to declare objects or in a component subtype definition.

Implementation Guideline: Include checks of generic formal array parameters and array types whose indexes are given by formal scalar types.

- T5. Check that the attributes give the appropriate values when applied to an unconstrained formal parameter. In particular, check that when the actual parameter is a slice (including a null slice), the appropriate results are produced.

- T6. Check that

- assignment (see IG 5.2.1/T1-T4),
- aggregates (for non-limited types) (see IG 4.3.2/T1-T23),
- membership tests (see IG 4.5.2.f/T65),
- qualification (see IG 4.7/T2),
- explicit conversion (see IG 4.6/T4), and

- indexing (implicit in various tests)

are defined for array objects.

- T7. Check that aggregates (see IG 4.3/T1), assignment, equality, and inequality (see IG 7.4.4/T4), and catenation (see IG 4.5.3.d/T41) are not defined for limited array types.

Check that slices are not defined for multidimensional array types (see IG 4.1.2/T1).

Check that ordering operations are not defined for one-dimensional arrays whose components have a fixed or floating point type (see IG 4.5.2.f/T61).

- T8. Check that slices are defined for one-dimensional arrays, even arrays of a limited type (see 4.1.2/T3).
- T9. Check that string literals are defined for one-dimensional arrays with character component types (see IG 3.6.3/T6).
- T10. Check that catenation is allowed for nonlimited, one-dimensional array types (see IG 4.5.3.d/T42-T47).
- T11. Check that not and logical operators are defined for one-dimensional arrays whose components have a boolean type (see IG 4.5.1.b/T11-T14).
- T12. Check that the attributes 'BASE and 'SIZE are defined for array types and subtypes (see IG 13.7.2/T3).
- T13. Check that the attributes 'SIZE and 'ADDRESS are defined for array objects (see IG 13.7.2/T3 and /T6).
- T14. Check that the catenation, string literal formation, and aggregate formation operations are declared for anonymous array types, so that these operations can be used to initialize an object.
- Check that string literal formation operators are not declared for multidimensional array types (see IG 3.6.3/T7).
- T15. Check that CONSTRAINT_ERROR is raised if the prefix of any array attribute has the value null (see IG 4.1.4/T1).
- T16. Check that T'RANGE is not allowed when T denotes an enumeration or integer type.

3.6.3 The Type String

Semantic Ramifications

S1. Because string literals are basic operations whose bounds are determined according to the rules for a positional aggregate (RM 4.2/3), they are implicitly indexed starting with POSITIVE FIRST, e.g., given

```
procedure P (X : STRING);
```

if we call P with the argument "ABC", then X'FIRST = 1 and X'LAST = 3. Similarly, if we initialize a constant with a string literal:

```
C: constant STRING := "ABC";
```

C'FIRST = 1 and C'LAST = 3.

S2. Any one-dimensional array with a discrete component type has string-like properties, since catenation and lexicographic ordering relations are predefined for such types. In addition, for

one-dimensional arrays of an enumeration type containing at least one character literal, string literals are defined using the characters in the set (see also IG 4.2/S).

S3. The rule for determining the upper bound of null string literals ensures that the upper bound is the predecessor of the lower bound, e.g.:

```
subtype NULL_STRING is STRING (5 .. 2);
procedure P (S : NULL_STRING);
...
P ("");          -- CONSTRAINT_ERROR raised
```

The lower bound of the null string literal is 5 and the upper bound is 4. The call P("") raises CONSTRAINT_ERROR because the bounds of the actual parameter do not equal those of the formal parameter.

S4. String literals are allowed in multidimensional array aggregates (RM 4.3.2/2) even though no string literal formation operation is declared for a multidimensional array type. For example,

```
type ARR is array (1 .. 3, 4 .. 5) of CHARACTER;
A : ARR := ("AB", "CD", "EF");          -- legal use of string literals
```

The declaration of ARR is *not* equivalent to:

```
type one_dim is array (4..5) of CHARACTER;
type two_dim is array (1..3) of one_dim;
```

No one-dimensional array is implicitly declared, since even after ARR is declared, the following expression is unambiguous:

```
"AB" = "AB"          -- unambiguous
```

This is unambiguous because only one equality operation is visible for a one-dimensional array of character type, namely, the operation declared in STANDARD for the type STRING. (Note: ('A', 'B') = ('A', 'B') would be ambiguous, however, because two equality operators are visible for composite types, and so the type of the aggregate ('A', 'B') cannot be determined from the context of its use.)

Changes from July 1982

S5. There are no significant changes.

Changes from July 1980

S6. The index subtype for predefined type STRING is now called POSITIVE.

Legality Rules

- L1. The type of a string literal must be determinable solely from the context in which the literal appears, but using the fact that the literal is a value of a one-dimensional array type whose component type is a character type (RM 4.2/4).
- L2. The character literals corresponding to the graphic characters contained within a string literal must be visible at the place where the string literal appears (RM 4.2/5).

Exception Conditions

- E1. CONSTRAINT_ERROR is raised for a null string literal if the lower bound is 'FIRST' of the index *base* type (RM 4.2/3).
- E2. CONSTRAINT_ERROR is raised for a string literal if any character in the literal does not belong to the component subtype (RM 4.3.2/11).

Test Objectives and Design Guidelines

- T1. Check that the predefined STRING type conforms to the specified definition; i.e., that POSITIVE'FIRST = 1 (except when the context dictates otherwise), POSITIVE'LAST = INTEGER'LAST, a null STRING literal has a lower bound of 1, and STRING subtypes can be declared with the upper bound INTEGER'LAST.
- T2. Check that a STRING variable can be declared with a lower bound value greater than one, e.g., 5 .. 10 or INTEGER'LAST - 5 .. INTEGER'LAST.
- T3. Check that the lower and upper bounds of null and non-null string literals are determined correctly (see IG 4.2/T5 and IG 4.3.2/T14).
Check that a string literal can be written whose upper bound is SYSTEM.MAX_INT or INTEGER'LAST.
- T4. Check that constant STRING objects can be declared whose bounds are determined by the initial value.
Implementation Guideline: Include strings formed by catenation.
- T5. Check that a STRING variable is considered an array, i.e., the array attributes can be applied to such a variable.
- T6. Check that string literals can be formed for character and string types other than the predefined CHARACTER and STRING type, and that the capabilities tested in the previous tests are available for such user-defined types (see IG 4.2/T4 and /T6).
- T7. Check that the declaration of a multidimensional array of character type does not implicitly declare a string literal formation operation.

3.7 Record Types

Semantic Ramifications

- S1. An array declared as a record component need not have static bounds or bounds specified with the name of a discriminant:

```

X : INTEGER := 5;
subtype S is INTEGER range 1..X;
type AR is array (S) of INTEGER;
type T is
  record
    A : STRING (1..X);  -- legal
    B : AR;              -- legal
  end record;

```

- S2. The name denoting a record component (other than a discriminant) cannot appear in the record definition in which the component was declared:

```

X : INTEGER := 20;
type T is
  record
    A : STRING (1 .. X);
    B : STRING (A'FIRST .. X);  -- illegal use of A
    C : INTEGER := A'FIRST;     -- illegal use of A
  end record;

```

The component declaration of B is illegal because its lower bound depends on a nondiscriminant component of the record. Similarly, the initial value of C contains an illegal use of A.

S3. RM 3.7/3 says that "Identifiers of all components of a record type must be distinct." Since a discriminant is a record component (RM 3.7.1/1), this means that names of discriminants must be distinct from each other and from the names of other components of a record. In addition, even though only one variant part can actually be present in any variant record value, the requirement for unique identifiers applies to the identifiers for every variant of a record, i.e., components belonging to different variants of a record type must have distinct identifiers, and the identifiers must also be distinct from those used for discriminants and nonvariant components (see IG 3.7.3/S for examples).

S4. The initialization expression for a component declaration is evaluated when an object that requires the default value is declared:

```

subtype S_3_4 is INTEGER range 3..4;
type R (M : S_3_4) is
  record
    case M is
      when 3 =>
        S : INTEGER range 1 .. 10;
      when 4 =>
        T : INTEGER range 11 .. 20;
    end case;
  end record;

R3_10 : constant R := (M => 3, S => 10);

type T (L : INTEGER := 3) is
  record
    DATA : STRING (1 .. L) := "ABC";
    DATUM : R(L) := R3_10;
  end record;

A : T(3);           -- default initial values are ok
B : T(4) := (L => 4, DATA => "ABCD",
             DATUM => (M => 4, T => 11));
C : T(4);           -- CONSTRAINT_ERROR raised
D : T;              -- default initial values are ok

```

Only the declaration of C raises CONSTRAINT_ERROR, since only in this case is an incompatible default value computed for DATA and DATUM: in DATA's case, the initial value is too short, and in DATUM's case, the discriminant value of the initial value is not 4. (The rule that says CONSTRAINT_ERROR is raised for C's declaration is given in RM 3.2.1/16 -- "The initialization of an object checks that the initial value belongs to the subtype of the object; ... the exception CONSTRAINT_ERROR is raised if this check fails.") CONSTRAINT_ERROR is not raised for B's declaration since RM 3.2.1/6 says default expressions are not used if an explicit initialization is given.

S5. For default values of array components, CONSTRAINT_ERROR is raised when the default value does not belong to the subtype of the component. Since the component is necessarily constrained (RM 3.6.1/6), CONSTRAINT_ERROR must be raised when the bounds of the component and the value are not the same; no "sliding" of bounds occurs. RM 3.2.1/16 states, "for an array object declared by an object declaration, an implicit subtype conversion is first

applied, as for assignment." A component declaration is not an object_declaration, and so the first part of the first sentence of RM 3.2.1/16 applies: "The initialization of an object (the declared object or one of its subcomponents) checks that the actual value belongs to the subtype of the object." For array components, this means the index values must be identical:

```

type T is array (1 .. 5) of INTEGER;
type R is
  record
    A : T := (2 .. 6 => 0);
  end record;
W : R;                                -- CONSTRAINT_ERROR raised
X : T := (2 .. 6 => 0);                -- no CONSTRAINT_ERROR raised
Y : R := (A => (2 .. 6 => 0));          -- CONSTRAINT_ERROR raised

```

CONSTRAINT_ERROR is raised for W because the default initialization value does not have the bounds 1..5. No exception is raised for X's declaration because X is an array declared by an object declaration, so a subtype conversion is performed before checking that the value belongs to X's subtype. The aggregate in Y's initialization value raises CONSTRAINT_ERROR because the bounds of A's value must equal the bounds specified in R's declaration (RM 4.3.2/11; "a check is made ... that the value of each subcomponent of the aggregate belongs to the subtype of this subcomponent").

S6. The definition of *null records* implies that any record with a discriminant is a non-null record, even if such a record has a null component list, e.g.:

```

type R (L : INTEGER) is
  record
    null;
  end record;

```

Objects of type R are not null records.

Changes from July 1982

S7. There are no significant changes.

Changes from July 1980

S8. A record must have at least one component (other than a discriminant) unless the component list contains the word "null."

S9. A component declaration can no longer contain an array type definition.

S10. It is explicitly stated that identifiers denoting the components of a record type must be distinct within the record declaration.

S11. The use of a name that denotes a component other than a discriminant is not allowed within the record type definition that declares the component.

S12. A component declaration with several identifiers is equivalent to a sequence of single component declarations.

S13. The default expressions for record components are no longer evaluated when the record type definition is elaborated.

S14. If the component list is null but the record has discriminants, then the record is not a null record.

Legality Rules

- L1. No duplicates are permitted among the identifiers declared in a set of component-declarations (RM 3.7/3). The set includes identifiers declared as discriminants (RM 3.7.1/1 and RM 3.7/3).
- L2. An array subtype indication in a component declaration must specify a constrained array (RM 3.6.1/6).
- L3. No dependencies between record components are permitted except for the use of a discriminant (RM 3.7/3) to specify: a bound in an index constraint, a discriminant value in a discriminant constraint, the discriminant governing a variant part, or an initialization expression (RM 3.7.1/6).
- L4. The base type of a record component and its initial value must be the same (RM 3.7/5).
- L5. A default initial value must not be specified for a component if assignment is not declared for the component's type, namely, if the component has a task type, a limited private type, or a composite type for which assignment is not declared (RM 3.7/5).

Exception Conditions

- E1. **CONSTRAINT_ERROR** is raised by the declaration of a variable in an object declaration if no explicit initial value is given in the object declaration, a default initial value is specified for a component having:
 - a scalar type, and the default value does not lie in the range specified for the component.
 - an array type, and the default value has bounds that do not equal those specified for the component.
 - a constrained type with discriminants, and the discriminants of the default value do not equal the discriminants specified for the component.
 - a constrained access type, and if the default value is not null, the discriminants or bounds of the object designated by the default value do not equal those specified for the component's subtype.
- E2. **CONSTRAINT_ERROR** is raised by the declaration of a variable in an object declaration if no explicit value is given in the object declaration and a default discriminant value is incompatible with its use within the record object (see IG 3.7.2/E for further details).

Test Objectives and Design Guidelines

- T1. Check that within a record type definition, duplicate record component identifiers are not permitted, either within a component declaration, between the component declarations preceding a variant part, within a variant part, between variant parts, between a variant part and any preceding component declaration, or between the names of discriminants and any record component name (see IG 8.3.c/T1).
- T2. Check that index constraints with nonstatic expressions can be used to constrain components of an array type.
- T3. Check that multiple component declarations are treated as a series of single component declarations, i.e., the components all have the same type and any expression used in constraints or for initialization is evaluated once for each component.
Implementation Guideline: Check for scalar, array, record, and access types. Consult IG 3.2/S for examples.
- T4. Check that

- an unconstrained array type cannot be used as a component type (even if it is initialized with a static value);
- the name of a nondiscriminant component of the record being defined cannot be used in specifying an index constraint, discriminant constraint, accuracy constraint, range constraint, or initial value of another component of the record;
- the name of the component being declared cannot be used to specify any constraint in its own subtype indication nor in its initial value (unless the identifier is used to denote another entity, e.g., a component of a record object);
- the name of the record type being declared cannot be used within the record to form an attribute of the record or a component of the record;
- the base types of a component and its initial value cannot be different;
- a default initial value cannot be specified for a component of a task type, a limited private type, or a composite type having a component for which assignment is not declared;
- a vacuous component_list is forbidden;
Implementation Guideline: Include checks for variant parts.
- array type definitions are not allowed in record component declarations.

T5. Check that a nonstatic scalar expression can be used to specify a component's range constraint or default initial value.

T6. For a component of a record, access, or private type, check that a nonstatic expression can be used in a discriminant constraint or in specifying a default initial value.

Check that for a component of a limited private type, a nonstatic expression can be used in a discriminant constraint.

Implementation Guideline: The types with discriminants should have different sizes for different values of the discriminants.

T7. For a component of an array type, check that a nonstatic expression can be used to specify its index constraint or in an aggregate specifying an initial value for the component.

T8. Check that CONSTRAINT_ERROR is raised when a record object without an explicit initialization is declared, if an incompatible value is specified as the default initial value for a subcomponent having a scalar, record, array, access, or private type.

Implementation Guideline: Check that the exception is not raised when the declaration is elaborated.

Implementation Guideline: Check also for components with constraints that depend on a discriminant.

T9. Check that an unconstrained record type can be used to declare a record component that can be initialized with an appropriate explicit or default value.

Implementation Guideline: Check also for values having components that depend on a discriminant.

T10. Check that expressions in constraints of component declarations are evaluated in the order the component declarations appear.

Check that expressions in an index constraint or discriminant constraint are evaluated when the component declaration is elaborated even if some bounds or discriminants are given by a discriminant of an enclosing record type.

T12. Check that a record can be declared without a variant part, with only a variant part, and with null.

3.7.1 Discriminants

Semantic Ramifications

S1. The values of the discriminants of a record object cannot be changed, even by a complete record assignment, if the object's declaration imposes a constraint (see IG 3.7.2/S).

S2. Both limited and nonlimited private types can be declared with discriminants.

S3. The role of default discriminants is discussed in IG 3.7.2/S.

S4. Discriminant declarations occur before those for any other components of a record type (see RM 3.3.1/2). This occurrence defines the order in which discriminant values must appear in positional record aggregates, namely, the discriminant values must be given first. (There is no requirement, however, that discriminants in a record representation occupy the initial locations of the record.) A discriminant can be given even if it is not used in any component of the record, e.g., even if there is no variant part.

S5. Although a name denoting a discriminant cannot be used in the default expression for another discriminant, such a discriminant name can be used in the default expressions for nondiscriminant components:

```

D2 : INTEGER := 5;
type R (D1 : INTEGER := D2;          -- ok      (1)
        D2 : INTEGER := D1;          -- illegal (2)
        D3 : INTEGER := D2) is      -- illegal (3)
  record
    C1 : INTEGER := D2;              -- ok      (4)
  end record;

```

The use of D2 at (1) is legal because it refers to the variable D2 declared outside the record declaration. The use of D1 at (2) is an illegal use of the (previously declared) discriminant D1. The D2 at (3) denotes the preceding discriminant according to the visibility rules and so cannot be used in D3's default initialization expression. The use of discriminant D2 at (4) is legal since C1 is not a discriminant component.

S6. A discriminant declaration may use the identifier of a discriminant declared earlier in the discriminant part as long as the identifier does not refer to the discriminant, i.e., the identifier may be used as a selector in a component selection, as a component simple name in an aggregate, or as a parameter name in a named parameter association:

```

package P is
  type F is
    record
      F : INTEGER;
    end record;
  subtype FF is F;
  G : F;
  function FUNC (F : INTEGER) return INTEGER;

  type Q (F : INTEGER;
          A : P.F;                      -- legal use of F
          H : FF := (F => 3);           -- legal use of F
          I : INTEGER := FUNC(F => 3);  -- legal use of F
          J : INTEGER := G.F) is      -- legal use of F

```



```

        record
            null;
        end record;

    end P;

```

S7. The rule limiting the complexity of expressions containing names of discriminants (RM 3.7.1/6) applies only to component subtype definitions. In particular, the rule does not apply to default initialization expressions of components:

```

type R (D : INTEGER) is
    record
        C : INTEGER := 2*D + D'SIZE;
    end record;

```

C's initialization expression contains legal uses of discriminant D.

S8. The rule in RM 3.7.1/6 does forbid the use of discriminant names in scalar constraints for components:

```

type R (D : INTEGER) is
    record
        C1 : INTEGER range D .. 3;           -- illegal
        C2 : FLOAT digits 3 range FLOAT(D) .. 3.0; -- illegal
        C3 : INTEGER range 1 .. F(new STRING(1..D)); -- illegal
    end record;

```

C3 is illegal even though the discriminant appears by itself in an index constraint.

S9. Since discriminants are components of a record (RM 3.7.1/1), and since all components must have distinct identifiers (RM 3.7/3), all discriminants must have distinct names and these names must also be different from names of other components of the record.

S10. Constraint expressions are evaluated when the record type definition is elaborated, except for those expressions consisting solely of discriminants of the enclosing record type:

```

type R (D : INTEGER) is
    record
        C1 : STRING (F1 .. D);
        C2 : REC (F1, D);
        C3 : INTEGER range F1..F2;
    end record;

```

The functions F1 and F2 are evaluated when R is elaborated (once for each occurrence of the function name).

S11. The meaning of RM 3.7.1/10, "The elaboration of a discriminant part has no other effect," is given in RM 3.1/9.

Changes from July 1982

S12. Identifiers of discriminants may be used within a record type definition if they do not denote discriminants of the record type.

Changes from July 1980

S13. The type of the discriminant must be specified by a type mark.

S14. A discriminant name may appear as the default expression for record components (other than discriminant components).

Legality Rules

- L1. The type of each discriminant must be an enumeration or integer type (RM 3.7.1/1).
- L2. The default initial value of the discriminant must have the same type as the discriminant (RM 3.7.1/4).
- L3. Default initial values must be provided for all or for none of the discriminants in a discriminant_part (RM 3.7.1/4).
- L4. The default expression of a discriminant must not contain a name denoting a discriminant declared earlier in the same discriminant part (RM 3.7.1/5).
- L5. The name denoting a discriminant of the record type being declared cannot be used in a range constraint, a floating point constraint, or a fixed point constraint of a component declaration (RM 3.7.1/6).
- L6. If a name denoting a discriminant of the record type being declared is used as a bound in an index constraint for a component, or as a discriminant value, the name must appear by itself as the only constituent of the expression (RM 3.7.1/6).
- L7. The names specified for discriminants must all be different, and none of the record component names specified in a record type definition can be the same as a discriminant name (RM 3.7/3).

Exception Conditions

- E1. CONSTRAINT_ERROR is raised when an object of a record type is declared without an explicit discriminant constraint and a default initial value for a discriminant lies outside the range of the discriminant's subtype.

Test Objectives and Design Guidelines

T1. Check that

- discriminants cannot have a fixed or floating point type, nor a composite, private, limited private, access, or task type;

Implementation Guideline: For the private and limited private cases, be sure that the private type is implemented as a discrete type.

- default initial values cannot be provided for only some discriminants;
- a discriminant of the type being declared is not allowed in an expression for a range constraint, a fixed point constraint, or a floating point constraint;

Implementation Guideline: Include a case like range 1 .. F(new T(1..DISC)).

Implementation Guideline: Check that the use of discriminants belonging to another record type is okay.

- the names of discriminants in a discriminant_part must all be different and cannot be the same as the names of any of the record components in the following record type definition (see IG 8.3.c/T1);
- direct assignments to discriminant components are forbidden (see IG 5.2/T2);
- a discriminant component cannot be named as an actual in out or out subprogram parameter (see IG 6.4.1/T1);
- a discriminant component cannot be named as an actual generic in out parameter (see IG 12.3.1/T2).

- T2. Check that an expression containing more than just the name of a discriminant cannot be used to specify the bound of an index or the value of a discriminant.

Implementation Guideline: Check attributes, e.g., D'SIZE and the use of a name by itself in a contained expression, e.g., A(DISC) or 1..F(new T(1..DISC)).

Check that a discriminant can be used in a complex default initial expression for a record component.

Check that a discriminant name may not be used as the default initial value for another discriminant declared later in the same discriminant part.

Check that the identifier for a discriminant can be used as a selected component (R.D) in an index or a discriminant constraint, as the name of a discriminant in a discriminant specification (D => 1), and as the choice in a function call (F(D => 1)) in a discriminant or index constraint.

T3. Check that the following types are permitted as the type of a discriminant:

- BOOLEAN,
- CHARACTER,
- user-defined enumeration type,
- all predefined integer types, and
- user-defined types derived from these types.

Implementation Guideline: Use an integer type, an enumeration type, and a derived discrete type in forming the derived types for this test.

T4. Check that a discriminant's default initial value must have the same type as the discriminant.

T5. Check that a record consisting only of discriminant components can be declared.

T6. Check that the type of a discriminant must be specified by a type_mark, not a subtype indication with a range constraint.

T7. Check that a default discriminant expression need not be static and is evaluated only when needed.

T8. Check that CONSTRAINT_ERROR is raised in an object declaration if a default initial value has been specified which violates the constraints of a component of a record or an array type whose constraint depends on a discriminant, and no explicit initialization is provided for the object.

Implementation Guideline: Check that CONSTRAINT_ERROR is not raised at the point of the declaration if a default initial value does not belong to the discriminant's subtype.

Check that CONSTRAINT_ERROR is not raised in the above case if the default initial value satisfies the constraint associated with the value of the discriminant given, either explicitly or by default in the object declaration.

Check that CONSTRAINT_ERROR is not raised when the record type definition is elaborated if the default value does not satisfy a component's constraint.

T9. Check that the only way to change the value of a discriminant is by whole record assignment, and this whole record assignment alters the constraint of components which depend on the discriminants (see IG 5.2/T1).

T10. Check that expressions in index constraints and discriminant constraints are evaluated when the record type definition is elaborated (except for expressions consisting solely of a reference to a discriminant of the type) (see IG 3.7/T10).

3.7.2 Discriminant Constraints

Semantic Ramifications

S1. For a type *T* denoting an unconstrained record, private, or limited private type declared with discriminants having no default initial values, the following contexts are the only ones in which *T* is permitted as a type mark without a discriminant constraint:

- a subtype declaration;
- a derived type definition (except for the full declaration of a private or incomplete type with discriminants; see RM 3.7.1/3 and RM 7.4.1/3);
- a conversion or qualified expression;
- a formal parameter declaration of a subprogram or generic unit;
- as the second operand of a membership operation (e.g., *X* in *T*; this is always TRUE);
- an access type definition;
- as an actual generic parameter corresponding to a generic formal type declaration that has no discriminant part or that has a corresponding discriminant part.

The contexts in which *T* requires a discriminant constraint in a subtype indication are:

- an object declaration that declares a variable (a discriminant constraint can be omitted in a constant declaration);
- a derived type definition for a private or incomplete type declared without discriminants.
- an array type definition (to specify the component type);
- a record component declaration;
- an allocator when the object being allocated is to be uninitialized except for its discriminants.

S2. If a discriminant part has no default initial values, all objects of that type are created with fixed discriminant values that cannot be changed by assignment. The presence of default discriminant values means that objects can be created whose discriminant values are not fixed; they can be changed by assignment to the entire object.

S3. The language rules ensure that the discriminants of every record object always have defined values. These values are provided either:

- explicitly by a discriminant constraint in a subtype declaration, object declaration, component declaration, or allocator,
- by an actual parameter (for subprogram and generic formal parameters), or
- in the absence of a discriminant constraint, by an initialization expression for the object or component, or, in the absence of an initialization expression, by the default expression for a discriminant.

S4. The default expression for a discriminant is not evaluated when a component subtype definition is elaborated if a value is provided by an initialization expression, e.g.:

```

type R1 (D : INTEGER := FUNC) is -- default expression is a function
  record
    C1 : INTEGER;
  end record;

X1 : R1 := (3, 3);                -- FUNC is not called here

type R2 is
  record
    C2 : R1;                      -- FUNC is not called here
    D : R1 := (4, 4);            -- FUNC is not called here
  end record;

X2 : R2 := ((5, 5), (6, 6));      -- FUNC is not called here
X3 : R2;                          -- FUNC is called once, for X3.C2

```

The default discriminant expression is only evaluated when its value is needed, i.e., when its value is not provided explicitly. FUNC is not called in the declaration of X1 since an explicit initialization expression is provided. Similarly, FUNC is not called in the declaration of component C2 since elaboration of a subtype indication that does not have a constraint means no default discriminant expression be evaluated (RM 3.3.2/6-8). FUNC is not called for the declaration of X2 since an explicit initialization expression is given (AI-00014). Finally, in the declaration of X3, FUNC is called to determine the default discriminant value for component X3.C2, but is not called for component X3.D since a default expression is given for this component (AI-00014).

S5. A default discriminant expression is also not evaluated for a component that does not belong to a particular record subtype, e.g.:

```

type R3 (D : INTEGER) is
  record
    case D is
      when 0 .. INTEGER'LAST =>
        C3 : R1;
      when others =>
        C4 : INTEGER;
    end case;
  end record;

X4 : R3(1);                      -- FUNC is called here
X5 : R3(-1);                     -- FUNC is not called

```

When elaborating X4's declaration, R3's discriminant value implies component X4.C3 exists, and so a discriminant value must be determined (and checked for compatibility); hence, FUNC is called. Since component X5.C3 does not exist, FUNC is not called when X5's declaration is elaborated.

S6. If R3 had a default discriminant and X4 was declared without a discriminant constraint, then the default discriminant value would determine the subtype of X4's value, and consequently, whether FUNC needs to be called.

S7. The value of the 'CONSTRAINED attribute for an object (see RM 3.7.4/3) tells whether objects have discriminant values that can be changed. This attribute is particularly relevant when assigning to unconstrained formal parameters, e.g.,

```

type T (L : INTEGER := 0) is
  record
    ...
  end record;

```

```

CONS_1, CONS_2 : T(5);
UNCONS_1, UNCONS_2 : T;

```

```

procedure P (X : in out T) is
begin
  X := UNCONS_1;
end P;

```

If we call P(CONS_1), the assignment to X raises CONSTRAINT_ERROR if the value of UNCONS_1.L is not 5; since CONS_1 is constrained, CONS_1 cannot be assigned a value that would change its discriminant. If we call P (UNCONS_2), then CONSTRAINT_ERROR will not be raised by the assignment to X, since the corresponding actual parameter does not have a fixed discriminant value.

S8. In the call P(CONS_1), X'CONSTRAINED is TRUE, implying that no assignment can be made to the actual parameter that would change its discriminant value. In the call P(UNCONS_1), X'CONSTRAINED is FALSE, and hence no check need be made to see whether the discriminant of the actual parameter is being changed.

S9. RM 3.7.2/5 defines how to check a discriminant value for compatibility. Two checks are required: 1) a check that the value belongs to the discriminant subtype, and 2) an "additional" check that the discriminant value is compatible with the discriminant's use in the subtype indications of each component that depends on the discriminant. For example:

```

type REC (D : INTEGER) is
  record
    C : STRING (D..3);
  end record;
subtype REC1 is REC(-1);           -- CONSTRAINT_ERROR
OBJ : REC(-1);                     -- CONSTRAINT_ERROR
type A1_REC is access REC(-1);     -- CONSTRAINT_ERROR

```

In each of these cases, when -1 is substituted for the discriminant in component C of the REC type, CONSTRAINT_ERROR is raised because -1 does not belong to STRING's index subtype. Similar examples can be constructed for components with discriminant constraints.

S10. When the type being constrained has a variant part, the "additional" check is performed only for those subcomponents that both exist for the subtype and that have a subtype definition that depends on a discriminant (AI-00358). For example, consider:

```

type ENUM is (A, B, C);
subtype ENUM_A is ENUM range A..A;
subtype ENUM_AB is ENUM range A..B;

type REC1 (D_A : ENUM_A; POS : POSITIVE) is
  record
    C1 : INTEGER;
  end record;

```

```

type REC2 (D_AB : ENUM_AB) is
  record
    case D_AB is
      when A =>
        C2 : REC1 (D_AB, 1);
        C3 : REC1 (D_AB, 0);      -- no CONSTRAINT_ERROR
        C4 : REC1 (A, 0);        -- CONSTRAINT_ERROR
      when B =>
        C5 : INTEGER;
    end case;
  end record;

OBJ_B : REC2 (B);                -- no CONSTRAINT_ERROR

```

CONSTRAINT_ERROR is not raised when the declaration of component C3 is elaborated because RM 3.7/8 says the component subtype definition is only elaborated if it does not depend on a discriminant. Since C3's subtype indication does depend on the discriminant, D_AB, no compatibility check is performed for any of C3's discriminant values. CONSTRAINT_ERROR will be raised for component C4, however, since this component's constraint does not depend on a discriminant and the value 0 does not belong to the subtype POSITIVE.

S11. If we assume that component C4 is eliminated from the type declaration, then the declaration of OBJ_B does not raise CONSTRAINT_ERROR; component C3 does not exist for OBJ_B, so its discriminant constraint is not checked for compatibility. On the other hand, if OBJ_B were declared as REC2(A), CONSTRAINT_ERROR would be raised since the value 0 does not belong to the subtype POSITIVE and component C3 exists for subtype REC2(A).

S12. A discriminant constraint can be given for private types and incomplete types with discriminants *before* the full declaration of the type. Since the full declaration has not yet been elaborated, the additional check required by RM 3.7.2/5 cannot always be performed when the discriminant constraint is elaborated:

```

package P is
  type INC (D1 : INTEGER);
  type ACC_INC is access INC(-1);      -- (1)
  type INC (D1 : INTEGER) is
    record
      S1 : STRING (D1 .. D1);
    end record;

  type PRIV (D2 : INTEGER) is private;
  subtype S2 is PRIV(-1);              -- (2)
private
  type PRIV (D2 : INTEGER) is
    record
      S2 : STRING (D2 .. D2);
    end record;

  type DEFER (D3 : INTEGER);
  type CONS_DEFER is access DEFER(-1); -- (3); illegal
  type ACC_DEFER is access DEFER;
  subtype SUB_ACC_DEFER is ACC_DEFER(-1); -- (4)
end P;

```

```

package body P is
  type DEFER (D3 : INTEGER) is

    record
      S3 : STRING (D3 .. D3);
    end record;
end P;

```

The wording in RM 3.7.2/5 appears to require that CONSTRAINT_ERROR be raised at points (1), (2), (3), and (4) even though the full record type declaration has not yet been elaborated at those points. In cases (3) and (4), the full type definition need not even be in the same compilation unit. The examples can be made more complicated to show that, in general, it is not possible to "look ahead" to the full type declaration, e.g.,

```

GLOBAL : INTEGER := -2;

function COMP return INTEGER is
begin
  return -GLOBAL;
end COMP;

package P is
  type T (D : INTEGER);
  subtype ST is T(-1);
  X : INTEGER range COMP .. 3;           -- 2 .. 3
  type T (D : INTEGER) is
    record
      S : STRING (D .. COMP);           -- -1 .. -2
    end record;
end P;

```

If compatibility of -1 as a discriminant value is checked against the full type declaration at the point where ST is declared, COMP will return the value 2, S's index constraint will be -1 .. 2, and CONSTRAINT_ERROR will be raised. But if the expressions in the declarations are elaborated in the proper order, COMP will have the value -2 when T's declaration is elaborated, and the range -1 .. -2 should not raise CONSTRAINT_ERROR.

S13. To resolve this difficulty, AI-00007 says:

- a discriminant constraint is not allowed in a subtype indication given in the private part of a package if the type mark denotes an incomplete type and the incomplete type's full declaration is given in the package body. (For example, the declaration at (3) is illegal according to this rule.)
- if a discriminant constraint is given for a type prior to the type's complete declaration (this can only happen for an incomplete type, a private type, or a type or subtype that has a subcomponent of an incompletely declared private type; see examples below), the additional check required by RM 3.7.2/5 is deferred and is performed no later than the end of the declaration that completely declares the type. (For example, CONSTRAINT_ERROR is not raised at points (1) or (2). Instead, the additional check required at (1) is performed no later than the end of INC's full declaration, and similarly, the check required at (2) is performed no later than PRIV's full declaration.)
- if a discriminant constraint is given for an access type, the additional check

required by RM 3.7.2/5 may, but need not, be performed. (For example, given that -1 belongs to INTEGER, no additional check need be performed because of the subtype indication given (4). No additional check is required even if DEFER's full declaration had been given prior to (4), and so no exception need be raised for (4).)

The rules established by AI-00007 resolve the difficulties posed by RM 3.7.2/5's requirement for an additional check, but do impose the additional implementation burden of saving the information needed to perform the additional check. For example, consider the following sequence of declarations:

```

package P is
  subtype INT7 is INTEGER range 1..7;
  subtype INT6 is INTEGER range 1..6;
  subtype INT5 is INTEGER range 1..5;

  type T_INT7 (D7 : INT7) is private;
  type T_INT6 (D6 : INT6) is private;
  type T_INT5 (D5 : INT5) is private;

  subtype T_CONS is T_INT7(6);          -- (1)

private
  type T_INT7 (D7 : INT7) is
    record
      C76 : T_INT6(D7);                -- (2)
    end record;

  type T_INT6 (D6 : INT6) is
    record
      C65 : T_INT5(D6);                -- (3)
      CF  : T_INT5(FUNC);              -- (4)
    end record;

  type T_INT5 (D5 : INT5) is
    record
      CF  : STRING (FUNC .. D5);       -- (5)
    end record;                        -- (6)

end P;
```

In this example, the full declaration of T_INT7 is not its complete declaration because the declaration of T_INT6 is not yet complete. Similarly, T_INT6's full declaration is not its complete declaration because of the use of T_INT5. T_INT5's full declaration is the complete declaration for T_INT5 and also completes the declaration of T_INT6 and T_INT7.

S14. CONSTRAINT_ERROR would be raised at point (1) if the discriminant's value were not in the range 1..7. Since it is in the required range, no exception is raised. However, additional checks are required to ensure that T_CONS's discriminant value is compatible with its use within T_INT7's declaration. These additional checks must be deferred until more information is available about T_INT7, but the checks cannot be deferred beyond the declaration that completes T_INT7, namely, the declaration of T_INT5.

S15. The declaration at (2) requires yet another deferred check since T_INT6 has not yet been fully (or completely) declared, and similarly, the declarations at (3) require deferred checks. At

the end of T_INT5's declaration, T_INT5, T_INT6, and T_INT7 have been completely declared and no additional deferred checks are required. AI-00007 requires that all deferred checks for types whose declaration is completed by T_INT5's declaration be performed no later than point (6). Since the value 6 is not compatible with D6's use at point 3, the checks deferred from point (2) and point (1) will require that CONSTRAINT_ERROR be raised. The exception can be raised at any point between point (3) (the first point at which a deferred check can fail) and point (6) (the point by which all deferred checks in this example must have been completed). (If the declaration at (3) had been

C65 : T_INT5 (6) :

CONSTRAINT_ERROR would have been raised immediately, since the first check required by RM 3.7.2/5 would fail.)

S16. From an implementation viewpoint, it is reasonable to require that the deferred checks be performed no later than point (6) since implementations must keep track of when types are completely declared (in order to enforce RM 7.4.1/4); keeping track of the deferred checks as well is likely to be only a slight additional burden. On the other hand, some implementations might find it easier to apply deferred checks on a step by step basis, e.g., checking at point (2) that the value 6 is compatible with D6's constraint and then checking at point (3) that 6 is compatible with D5's constraint. This approach is allowed by AI-00007. Requiring the deferred checks to be performed at any specific point earlier than point (6) might be inconvenient for some implementations, but being more specific about where the checks are performed is of only marginal value to a programmer, whose program is, in any case, in error.

S17. AI-00007 does not require a subcomponent compatibility check when a discriminant constraint is applied to an access type. Such checks can be difficult or impossible to perform correctly and they are not necessary (i.e., failure to perform the check does not allow an invalid object to be created).

S18. To see why it is safe to eliminate the subcomponent check, consider:

```
type ACC_STR is access STRING;
type VSTR (FIRST, LAST : INTEGER) is
  record
    DATA : ACC_STR (FIRST .. LAST);
  end record;
X : VSTR (-1, -1);           -- optional exception
```

CONSTRAINT_ERROR will be raised if a subcomponent check is performed for X.DATA's designated type, since -1 is not an allowed lower bound value for STRINGS. If no such check is performed, then an exception will be raised later:

```
X := (-1, -1, new STRING (-1..-1));  -- CONSTRAINT_ERROR raised (1)
X := (1, 3, new STRING (1..3));      -- CONSTRAINT_ERROR raised (2)
X := (-1, -1, new STRING' ("abc"));  -- CONSTRAINT_ERROR raised (3)
```

In case (1), an exception is raised by the allocator, since -1 is not an allowed lower bound. In case (2), the aggregate produces an allowed value of type VSTR, but an exception is raised by the assignment since X's discriminant constraint is not satisfied. In case (3), CONSTRAINT_ERROR is raised because the bounds of the designated object are not the same as the discriminant values.

S19. Now consider a somewhat more complex example:

```
type ACC_VSTR is access VSTR;
Y : ACC_VSTR (-1, -1);           -- no exception need be raised
```

An allocator creating a value to be assigned to Y need not raise an exception either:

```
Y := new VSTR (-1, -1);           -- no exception need be raised
```

This is acceptable since Y.DATA = null; no invalid object has been created. On the other hand, an implementation could raise `CONSTRAINT_ERROR` for the allocator if it performs the subcomponent check for the DATA component.

S20. Allowing the subcomponent check to be omitted simplifies an implementation considerably. For example, consider:

```
type R1 (D1 : INTEGER);
type R2 (D2 : INTEGER);
type R3 (D3 : POSITIVE);

type ACC_R1 is access R1;
type ACC_R2 is access R2;
type ACC_R3 is access R3;

type R1 (D1 : INTEGER) is
  record
    C1 : ACC_R2 (D1);
  end record;

X1 : R1 (-1);           -- can't do complete subcomponent check

type R2 (D2 : INTEGER) is
  record
    C2 : ACC_R3 (D2);
  end record;

type R3 (D3 : POSITIVE) is
  record
    C3 : ACC_R1 (D3);
  end record;
```

The declaration of X1 would be illegal if C1's type were an incomplete type (e.g., R2) or if it were a private type prior to its complete declaration. But since C1 has an access type, X1's declaration is legal. A complete subcomponent check can't be performed when X1 is declared since R2 has not yet been completely declared. Nonetheless, the object denoted by X1 can be created. One could even create a function (by generic instantiation) that could be called with X1 as an actual parameter.

S21. Performing a subcomponent check for an access type can be complex. Suppose the following declaration appears after R3's full declaration and an implementation attempts to perform the subcomponent check:

```
A1 : ACC_R1 (10);
```

When performing the check for A1, an implementation must be careful not to get into a loop, since both A1 and A1.C1.C2.C3 have the same access type. The complexity of avoiding a loop is increased when one considers that any of the designated types could be variant records, i.e., the choice of which subcomponents to check could depend on which subcomponents are present for particular discriminant values (see AI-00358). Since performing the subcomponent check for an access type is sometimes impossible and sometimes complex, it seems reasonable for an implementation to take advantage of AI-00007 and not perform the check at all.

S22. Of course, only the subcomponent check can be omitted:

```
A1 : ACC_R3 (-1);      -- raises CONSTRAINT_ERROR
```

CONSTRAINT_ERROR must be raised since -1 does not belong to R3's discriminant subtype. No subcomponent compatibility check is needed.

Approved Interpretations

S23. AI-00007 imposes the following rules:

- A discriminant constraint is not allowed in a subtype indication given in the private part of a package if the type mark denotes an incomplete type and the incomplete type's full declaration is given in the package body.
- If the type mark in a subtype indication denotes an incomplete type, a private type, or a type or subtype that has a subcomponent of an incompletely declared private type and if the subtype indication has a discriminant constraint and occurs prior to the end of the complete declaration of the type denoted by the type mark, then the compatibility of each discriminant value with its use in a component subtype definition is checked no later than the end of the declaration that completely declares the type. If this check fails, **CONSTRAINT_ERROR** is raised.
- If a subtype indication contains a discriminant constraint and the type mark denotes an access type, the discriminant constraint is compatible with the type denoted by the type mark if each discriminant value belongs to the subtype of the corresponding discriminant of the designated type. In addition, the compatibility of each discriminant value with its use in a component subtype definition of the designated type, may, but need not be checked. (If the additional check is performed and fails, **CONSTRAINT_ERROR** is raised.)
- (The complete declaration of an incomplete or private type is its full declaration if the full declaration declares a scalar type or an access type. If the full declaration declares a type with components, the full declaration is the complete declaration only if the type of each component has been completely declared; otherwise, the type is completely declared by the last complete declaration of a component's type.)

S24. AI-00014 requires that a default discriminant expression be evaluated (and checked for compatibility) only if this is the only source of a discriminant value.

S25. AI-00358 requires that the compatibility of each discriminant value with its use within a record only be checked for those subcomponents that exist for the record's subtype and whose subtype definition depends on a discriminant.

Changes from July 1982

S26. A discriminant association with more than one discriminant name is only allowed if discriminants are all of the same type.

S27. A declaration of a constant object with discriminants does not require a discriminant constraint in the object declaration even if there are no default values for the discriminants.

Changes from July 1980

S28. To check discriminant constraint compatibility for each subcomponent whose subtype depends on the discriminant, the discriminant value is substituted for the discriminant in the specification and the compatibility of the resulting subtype is checked.

S29. In the elaboration of discriminant constraints the expressions are elaborated in some order that is not defined by the language.

S30. The expression of a named association is evaluated once for each named discriminant.

Legality Rules

- L1. A discriminant constraint in a subtype indication can only be given for an unconstrained type with discriminants, or for an unconstrained access type designating an unconstrained type with discriminants (RM 3.7.2/1).
- L2. If a discriminant constraint contains both positional and named associations, the positional associations must be given first (RM 3.7.2/3).
- L3. A discriminant name in a discriminant constraint must only be the name of the discriminant of the type for which the constraint is being specified (RM 3.7.2/4).
- L4. A discriminant association with more than one discriminant name is only allowed for discriminants of the same type (RM 3.7.2/4).
- L5. The base type of a discriminant specified in the discriminant constraint and the corresponding discriminant in a record type must be the same (RM 3.7.2/4).
- L6. A discriminant constraint must specify exactly one value for each discriminant (RM 3.7.2/4).
- L7. For a record or private type declared without default discriminant values, a discriminant constraint must be specified in a subtype indication used in an object declaration to declare a variable, the declaration of an array's component type, the declaration of a record component, and an allocator of the form `new T` (RM 3.7.2/8, /10).
- L8. A discriminant constraint is not allowed in a subtype indication given in the private part of a package if the type mark in the subtype indication denotes an incomplete type and the incomplete type's full declaration is given in the package body (AI-00007).

Exception Conditions

- E1. `CONSTRAINT_ERROR` is raised when an object declaration or a discriminant constraint that does not depend on a discriminant is elaborated and the value specified for any discriminant (either by default or explicitly) does not lie in the permitted range of values for the discriminant (RM 3.3.2/9 and RM 3.7.2/5).
- E2. If the full declaration for a type with discriminants, `T`, contains a component subtype definition that is dependent on a discriminant (i.e., if the component subtype definition contains an index or a discriminant constraint that uses a discriminant of the enclosing type) and a subtype indication giving a discriminant constraint for `T` is elaborated before `T`'s complete declaration, `CONSTRAINT_ERROR` is raised no later than the end of `T`'s complete declaration if a specified discriminant value is not compatible with its use in specifying the component's subtype (RM 3.7.2/5 and AI-00007), i.e., `CONSTRAINT_ERROR` is raised if:
 - the discriminant is used in a discrete range of an index constraint, the range is not null, and at least one bound does not belong to the index subtype; or
 - the discriminant is used in a discriminant constraint and the value of any discriminant in the constraint does not lie in the range of values permitted for the discriminant.

In addition, `CONSTRAINT_ERROR` is raised no later than the end of `T`'s complete

declaration if some other discriminant value in such a component subtype definition is not compatible with the component type (RM 3.7.2/5 and AI-00007).

- E3. If the full declaration for a type with discriminants, T, contains a component subtype definition that is dependent on a discriminant (i.e., if the component subtype definition contains an index or a discriminant constraint that uses a discriminant of the enclosing type) and a subtype indication giving a discriminant constraint for T is elaborated after T's complete declaration, CONSTRAINT_ERROR is raised if a specified discriminant value is not compatible with its use in specifying the component's subtype (RM 3.7.2/5), i.e., CONSTRAINT_ERROR is raised if:

- the discriminant is used in a discrete range of an index constraint, the range is not null, and at least one bound does not belong to the index subtype; or
- the discriminant is used in a discriminant constraint and the value of any discriminant in the constraint does not lie in the range of values permitted for the discriminant.

In addition, CONSTRAINT_ERROR is raised if some other discriminant value in such a component subtype definition is not compatible with the component type (RM 3.7.2/5 and AI-00007).

- E4. If the type mark in a subtype indication denotes an access type and the subtype indication contains a discriminant constraint, CONSTRAINT_ERROR may, but need not be raised when the subtype indication is elaborated, if a specified discriminant value is incompatible with its use in a component subtype definition of the designated type (AI-00007 and RM 3.7.2/5). In addition, CONSTRAINT_ERROR may, but need not be raised if some other discriminant value in such a component subtype definition is not compatible with the component type (RM 3.7.2/5 and AI-00007).

Test Objectives and Design Guidelines

- T1. Check that the form of a discriminant constraint is correct; namely, check that:

- the discriminant names given in the constraint cannot be different from the names of the discriminants of the type being constrained;
- the same name cannot appear twice as a discriminant name in a particular discriminant association or in different discriminant associations of the same discriminant constraint;
- if a mixture of positional and named association is used, a named discriminant association cannot give a value for a discriminant whose value has already been specified positionally;

Implementation Guideline: Be sure that the total number of discriminant values does not exceed the number of discriminants being constrained.

- too many or too few discriminant values cannot be given;
- positional discriminant values cannot be given after a discriminant association that uses discriminant names;

Implementation Guideline: The total number of discriminant values should be correct; the positional values should be in the correct position and of the correct type.

- the base type of a discriminant value cannot be different from the base type of the corresponding discriminant;

Implementation Guideline: Use different discrete types.

- a discriminant association with more than one discriminant name is only allowed if the named discriminants are all of the same type.

Implementation Guideline: Include checks for access types that designate types with discriminants.

- T2. Check that discriminant constraints are not permitted where they are forbidden; i.e., check that a discriminant constraint:
- cannot be given in a subtype indication for a type mark that has already been constrained (see IG 3.3.2/T5);
 - cannot be specified for a record or private type declared without any discriminants, or for an array, scalar, or task type (see IG 3.3.2/T5).
 - cannot be supplied with an actual generic parameter (see IG 12.3/T5).
- T3. Check that a discriminant constraint cannot be omitted where it is required, i.e., check that a discriminant constraint cannot be omitted from a subtype indication for a type mark having discriminants with no default values when the subtype indication is used in:
- an object declaration for a variable;
 - an array type definition;
 - a record component declaration;
 - a derived type declaration used as the full declaration of a private or a limited private type that has no discriminants (see IG 7.4.1/T5);
 - an allocator when an initial value or discriminant constraint is not specified for the allocated object (see IG 4.8/T1).
- T5. Check that if a discriminant of type T is named A, then in a discriminant association for the type, T.A cannot be used as a name for the discriminant.
- T6. For a type without default discriminant values (but with discriminants), check that an unconstrained type mark can be used in:
- a subtype declaration, and the subtype name acts simply as a new name for the unconstrained type;
 - a derived type declaration, except when the declaration is the full declaration of a private or limited private type without discriminants (see IG 7.4.1/T5) or the full declaration of an incomplete type (see IG 3.8.1/T2);
 - an access type definition (see IG 3.8/T2) (hence a discriminant constraint must be supplied in an allocator that does not provide an initial value (see IG 4.8/T1));
 - a membership operation (see IG 4.5.2.a/T2);
 - a formal parameter declaration for a subprogram; hence, the constraints of the actual parameter are available within the subprogram, 'CONSTRAINED is TRUE, and assignments to the formal parameter cannot attempt to change the discriminants of the actual parameter without raising CONSTRAINT_ERROR (see IG 6.4.1/T6);
- Implementation Guideline:* For the actual parameter, use an object designated by an access value as well as other objects.
- as an actual generic parameter corresponding to a generic (private) type declaration having a discriminant part (see IG 12.3.2/T2);
 - the declaration of a constant.

T7. For a type with or without default discriminant values, check that a discriminant constraint can be supplied in the following contexts and has the proper effect:

- in an object declaration, component declaration, or subtype indication of an array type definition; hence, assignments cannot attempt to change the specified discriminant values without raising `CONSTRAINT_ERROR`;

Implementation Guideline: Check that the size of the object can depend on the value of a discriminant.

- in an access type definition; hence, access values of this access type cannot be assigned non-null values designating objects with different discriminant values (see IG 5.2/T9);

- in an allocator, and the allocated object has the specified discriminant values (see IG 4.8/T5).

T8. For a type with default discriminant values, check that a discriminant constraint can be omitted in:

- an object declaration; hence, assignments to the object can change its discriminants;
- a component declaration in a record type definition; hence, assignments to the component can change the value of its discriminants;
- a subtype indication in an array type definition; hence, assignments to one of the components can change their discriminant values;
- an allocator, but an assignment to the allocated object cannot change the default discriminant values (see IG 4.8/T4 and IG 4.8/T7);
- a formal parameter specification of a generic unit or subprogram; for in out or out parameters, 'CONSTRAINED of the actual parameter equals 'CONSTRAINED of the formal parameter. If 'CONSTRAINED is TRUE, `CONSTRAINT_ERROR` is raised when an attempt is made to change the value of a discriminant.

In each of the above cases, check that the default discriminant expression is checked for compatibility:

- in an object declaration, when no initialization expression or discriminant constraint is present in the declaration itself;
- for a subcomponent of an object, when no initialization expression is present in the object declaration nor is a default expression provided in the corresponding component subtype definition or in the component subtype definition of an enclosing component;

Implementation Guideline: Check that the expression is evaluated and checked if and only if the component is present in the record subtype. Include a case where the subtype is itself determined by a default discriminant value.

- never in a record component declaration, subtype declaration, array type definition, derived type definition, or access type definition.

Implementation Guideline: Include a check using formal generic parameters.

Implementation Guideline: Note: IG 3.2.1/T8 checks that default discriminants are evaluated at the appropriate time.

T9. Check that `CONSTRAINT_ERROR` is not raised for a constant object declaration whose

subtype indication specifies an unconstrained type with default discriminant values, and whose initialization expression specifies a value whose discriminants are not all equal to the default values.

Check that `CONSTRAINT_ERROR` is raised when the subtype indication in a constant object declaration specifies a constrained subtype with discriminants, and the initialization value does not belong to the subtype (i.e., the discriminant values do not match those specified by the constraint).

- T10. Check that the expression in a discriminant association with more than one name is evaluated once for each name.
- T11. Check that `CONSTRAINT_ERROR` is raised by a discriminant constraint if a value specified for a discriminant does not lie in the range of the discriminant and the discriminant constraint does not itself depend on a discriminant.

Implementation Guideline: Check for subtype indications where the type mark denotes a record, private, incomplete, and access type.

Implementation Guideline: Check for types whose full declaration occurs both before and after the use of the discriminant constraint.

Implementation Guideline: Check for subtype indications used in subtype declarations, array component declarations, record component declarations (including component declarations appearing in a variant part. Do not use constraints dependent on a discriminant; see T13), access type definitions, derived type definitions, object declarations, and allocators (for allocators, see IG 4.8/T8).

Implementation Guideline: Check that `CONSTRAINT_ERROR` is raised when the subtype indication is elaborated (as opposed to when the full type declaration is elaborated or when an object of the subtype is created).

- T12. For an incomplete type declared in the private part of a package, check that a discriminant constraint is not allowed for the type (in an access type definition) if the full type declaration does not also occur in the private part.
- T13. Check that if a discriminant constraint or index constraint depends on a discriminant, the non-discriminant expressions in the constraint are evaluated and checked at the appropriate time. Specifically, if a component subtype definition contains a constraint that depends on a discriminant, check that the expressions in the constraint are evaluated when the subtype definition is elaborated but the compatibility of discriminant values is not checked until later when the record type is either constrained explicitly (in a subtype declaration, array component declaration, record component declaration that does not depend on a discriminant, access type definition, derived type definition, object declaration, and allocator) or is used without a constraint in an allocator or object declaration that has no initialization expression; furthermore, ensure that compatibility is only checked if the component is present in the record subtype.

Implementation Guideline: Include a check for generic formal types.

Implementation Guideline: The constrained component should have a record, array, and access type.

Implementation Guideline: In some cases, the subcomponent should be in a variant part. The test should ensure that the compatibility of the discriminant values is checked if and only if the component is present in the subtype that is specified either by the explicit discriminant constraint or by the default discriminant values.

Implementation Guideline: This test is concerned with checking the compatibility of discriminant expressions that are not themselves discriminants. Checks of enclosing discriminant values are covered in T15 and T16.

Implementation Guideline: The discriminant constraints in this test should not involve incompletely declared types. Incompletely declared types are checked in T14.

- T14. Ensure that the compatibility checks tested in T13 are performed correctly when the type being constrained has not been completely declared, i.e., ensure that all compatibility checks are performed no later than the end of the declaration that completely declares a type.

Implementation Guideline: When a component's discriminant constraint is given for an incompletely declared type and depends on a discriminant, the compatibility check of all discriminant values can be deferred. This test should ensure the compatibility check is not deferred too long; see examples in IG 3.7.2.S.

T15. Repeat the checks of T13, but this time, ensure compatibility checks are performed for discriminant values that depend on discriminants.

T16. Repeat the checks of T14, using discriminant values that depend on discriminants.

T17. Check whether the additional compatibility check is performed when a discriminant constraint is given for an access type.

Implementation Guideline: Check when the designated type has been completely declared, is completely declared later (in the same or in a different compilation unit), and when there is a "loop" in the designated types to be checked (see IG 3.7.2/S).

Implementation Guideline: Include a check for access types used in a component subtype definition and in an allocator.

3.7.3 Variant Parts

Semantic Ramifications

S1. A choice can have the form of a component_simple_name only in a record aggregate. The syntactic alternative, component simple name, exists so record and array aggregates can be specified with the same syntactic rule (see RM 4.3/2).

S2. The subtype of the discriminant simple name in a variant part is given in the corresponding discriminant part. The subtype indication must be a type mark, T, which could have been declared in one of the following ways:

```

type T is range L .. R;           -- 1
type T is (...);                 -- 2
type T is new ST;                 -- 3
type T is new ST range L .. R;   -- 4
subtype T is ST;                  -- 5
subtype T is ST range L .. R;    -- 6

```

In form 1, L and R must be static (see RM 3.5.4/3), and so T is a static subtype. In form 2, T is a static subtype since it is an enumeration type. In forms 3 and 5, T is static if ST is static. In forms 4 and 6, T is static if L and R are both static and ST is static.

S3. For a type declaration like:

```

subtype INT is INTEGER range 1 .. 10;
type T (L : INT) is
  record
    case L is
      ...;
    end case;
  end record;

```

the set of choices must only cover the range 1 through 10. Values outside this range cannot be specified. For example,

```

case L is
  when 1 .. 10 =>
    A : INTEGER;
  when 11 .. 20 => -- illegal
    null;
end case;

```

is illegal even though this choice defines no components and the choice can never be chosen.

S4. Note that the following is legal:

```
case L is
  when 1 .. 10 =>
    A : INTEGER;
  when 20 .. 11 =>    -- legal
    B : BOOLEAN;
end case;
```

because the RM states "... no other value is allowed" (RM 3.7.3/3); 20 .. 11 is a null range which contains no values, so the null range is legal.

S5. If the subtype of the discriminant name is not static, then the base type of the discriminant specifies the range of values that must be covered.

S6. Since components of a record must have unique identifiers (RM 3.7/3), components of all variant parts must have distinct identifiers:

```
case L is
  when 1..5 =>
    A : INTEGER;
    B : FLOAT;
  when 6..7 =>
    A : INTEGER;    -- illegal
```

S7. A variant part can appear within a variant part, e.g.,

```
case L is
  when 1 .. 5 =>
    case M is
      when 1 .. 10 =>
        etc.
```

Even if the value of discriminant M is used only in a variant part when L is in the range 1 .. 5, space must be allocated for M within the record object no matter what the value of L is for a particular object. Discriminants are components of every object of a type with a discriminant (see RM 3.7.1/1).

S8. The same discriminant name can be used in more than one variant part:

```
case L is
  when 1 .. 5 =>
    case L is
      when 1 .. 2 =>
```

The set of choices to be covered in the inner variant part is the same as the set in the outer part. Covering just the choice values 1 .. 5 in the inner part would be illegal if the subtype of L was INTEGER range 1 .. 10.

Changes from July 1982

S9. The type of the discriminant of a variant part must not be a generic formal type.

Changes from July 1980

S10. If the subtype of a discriminant is static, the choices in a variant part must contain no values other than those of the subtype.

Legality Rules

- L1. The base type of the discriminant and each choice must be the same (RM 3.7.3/1).
- L2. The type of the discriminant of a variant part must not be a generic formal type (RM 3.7.3/3).
- L3. When the subtype of a discriminant is static, the choices in a variant part must contain all and only values of the subtype (RM 3.7.3/4).
- L4. If the subtype of a discriminant is not static, the choices in a variant part must contain all and only values of the discriminant's base type (RM 3.7.3/4).
- L5. Two choices must not have a value in common (RM 3.7.3/4).
- L6. Every choice must contain only static expressions, i.e., for choices of the form V, L .. R, and ST range L .. R, L, R, and V must be static expressions and ST must be a static subtype; for choices of the form ST, ST must be a static subtype (RM 3.7.3/4).
- L7. An **others** choice, if present, must be the only choice given in the last alternative specified for a variant part (RM 3.7.3/4).
- L8. An **others** choice must be present if the set of values included in the set of choices does not cover 1) the set of values associated with the subtype of the discriminant name when the subtype is static; otherwise, 2) the set of values associated with the base type of the discriminant (RM 3.7.3/4).
- L9. A component simple name is not allowed as a choice of a variant (RM 3.7.3/4).
- L10. Identifiers belonging to different variants of the same record type must have unique identifiers (RM 3.7/3).

Test Objectives and Design Guidelines

- T1. Check that (cf. IG 5.4.a/T1):
 - the reserved word **is** is required;
 - **when** cannot be replaced by **if**,
 | cannot be replaced by **or**,
 => cannot be replaced by **then**,
 end case cannot be replaced by **end**, **endcase**, or **esac**,
 is cannot be replaced by **of**;
 - the **others** choice must be the only choice given in the last alternative;
 Implementation Guideline: Try an **others** choice as the first and middle alternative, and try it as the first, middle, and last choice in a set of choices for the last alternative.
 - a component list cannot be vacuous.
- T2. Check that (cf. IG 5.4.a/T20):
 - the type of the discriminant and each choice must be the same;
 - every pair of choices must cover a disjoint set of values.

Implementation Guideline: Use both single values and ranges of values, and check for overlapping values with a single alternative and between alternatives. Use overlapping ranges whose end points are different, e.g., 3 .. 5 and 4 .. 6, as well as ranges in which overlap occurs only at the end points. Use some examples in which a large range of values has to be checked for potential overlap. These choices should not all occur in monotonically increasing or decreasing order.
- T3. Check that nonstatic choice values are forbidden (cf. IG 5.4.a/T21).

Implementation Guideline: Try a variable whose range is restricted to a single value. Try a discrete range of the form ST, where ST is a subtype name having at least one nonstatic bound, as well as choices of the form ST range L .. R and L .. R, where either L, R, or ST is nonstatic. Include all nonstatic forms of expressions, specifically a case where an exception would be raised, e.g., POSITIVE range 0 .. 3. Also try A RANGE.

- T4. Check that all forms of choice are permitted in variants, and in particular, forms like ST range L .. R, and ST are permitted (cf. IG 5.4.a/T22).

Implementation Guideline: Use the same subtype name in more than one choice.

Check that choices using named numbers and static constant names are permitted (cf. IG 5.4.a/T23).

- T5. Check that choices denoting a null range of values are permitted, and that for choices of the form ST range L .. R where $L > R$, neither L nor R need be in the range of ST values (cf. IG 5.4.a/T24).

Implementation Guideline: The alternative specified by a null choice should have null as its component list in one test and a non-null list in a separate test.

Check also that an others alternative can be provided even if all values of the case expression have been covered by preceding alternatives (cf. IG 5.4.a/T24).

- T6. Check that choices within and between alternatives can appear in nonmonotonic order (cf. IG 5.4.a/T26).
- T7. Check that relational, membership, and logical operators are allowed as choices only if the expressions containing these operators are enclosed in parentheses (cf. IG 5.4.a/T27).
- T9. Check that if a discriminant has a static subtype, an others choice can be omitted if all values in the subtype's range are covered, and must not be omitted if one or more of these values are missing (cf. IG 5.4.b/T1).

Check that values outside the range of the subtype are forbidden, even if the component list for such an alternative is null.

- T10. Check that if the subtype of a discriminant is not static, others can be omitted if all values in the base type's range are covered, and must not be omitted if one or more of these values are missing (cf. IG 5.4.b/T2).
- T11. Check that even when the context indicates that a discriminant covers a smaller range of values than permitted by its subtype, an others alternative is required if the subtype value range is not fully covered (cf. IG 5.4.b/T4).

Implementation Guideline: Use the nested variant part example at least.

- T12. Check that the type of the discriminant of a variant part must not be a generic formal type if the discriminant is used to govern a variant part. Check that a discriminant otherwise may have a generic formal discrete type.

Implementation Guideline: Use a generic formal type to constrain a component that appears within a variant.

3.7.4 Operations of Record Types

Semantic Ramifications

- S1. 'CONSTRAINED has the value TRUE for:

- an object or component declared with a discriminant constraint;
- an object designated by an access value, since no assignment to such an object is permitted to change its discriminants (see RM 4.8/5), whether or not the type of the object was declared with default discriminant values;

- a constrained formal subprogram, entry, or generic parameter (of any mode);
- an unconstrained formal subprogram, entry, or generic parameter of mode in out or out whose actual parameter's 'CONSTRAINED attribute is TRUE;
- a formal subprogram, entry, or generic parameter of mode in (regardless of the value of the actual parameter's 'CONSTRAINED attribute).

'CONSTRAINED has the value FALSE for:

- objects declared without a discriminant constraint in an object declaration or in a component of a record or array; such declarations are only legal for record, private, or limited private types with default discriminant values;
- an unconstrained formal subprogram, entry, or generic parameter of mode in out or out whose actual parameter's 'CONSTRAINED attribute is FALSE.

s2. 'CONSTRAINED is explicitly defined for parameters of mode out, even though reading of out parameters is sharply restricted (see IG 6.4.2/S and RM 6.4.2/5).

s3. 'CONSTRAINED is also defined for private types (see RM 7.4.2/9, 10), but not for objects of such types unless the private type has discriminants:

```
generic
    type T is private;
    procedure PR (X : T);

    procedure PR (X : T) is
        B1 : BOOLEAN := T'CONSTRAINED;    -- legal
        B2 : BOOLEAN := X'CONSTRAINED;    -- illegal
```

T'CONSTRAINED is legal because T is a private type. X'CONSTRAINED is illegal because T is not declared to have discriminants. Note that 'CONSTRAINED is not defined for record types that have discriminants.

s4. The prefix of 'CONSTRAINED must be an object (or a private type). Since functions deliver values, not objects, a function cannot be a prefix of this attribute even when the function returns an access value:

```
type R (D : INTEGER) is
    record
        C : INTEGER;
    end record;
type ACC_R is access R;

function F return R;
function G return ACC_R;

...F'CONSTRAINED    -- illegal
...G'CONSTRAINED    -- illegal
```

s5. There are several possible implementation approaches for obtaining the value of 'CONSTRAINED for a given object. If the object is not an unconstrained formal in out or out parameter, the attribute's value can be determined at compile time. If the object is an unconstrained in out or out formal parameter, the actual parameter's 'CONSTRAINED value could be passed as "dope" to the formal parameter; this is necessary only if the discriminants of the formal parameter's type have default discriminant values. The value of 'CONSTRAINED

could also be stored within the object itself, but since 'CONSTRAINED' is a property of an object, not its value, an implementation must then be careful not to copy the value of 'CONSTRAINED' when making assignments between two objects whose 'CONSTRAINED' attributes are potentially different, or when comparing the objects for equality. For example:

```

type T (L : INTEGER := 0) is
  record
    ...
  end record;

CONS_1   : T(5);
UNCONS_1 : T;

CONS_2   : T(5) := CONS_1;
UNCONS_2 : T    := UNCONS_1;

```

These initializations can copy the value of 'CONSTRAINED', since the value of 'CONSTRAINED' is the same for the variable being declared and the initial value. However, copying must *not* be done for:

```

CONS_2 := UNCONS_1;
UNCONS_2 := CONS_1;

```

The value of 'CONSTRAINED' must not be changed by assignment.

s6. The aggregate formation and component selection operations are not defined for null records. Null records have no components to select, and there is no form of aggregate for a null record. All other basic operations are, however, defined for null records. Specifically:

```

type NULL_REC is
  record
    null;
  end record;

R1, R2 : NULL_REC;
...
R1 := R2;           -- ok

if R1 = R2 then     -- always TRUE
  ...
end if;

```

The assignment statement and comparison expression are legal. Note that evaluation of R1 and R2 is not erroneous even though R1 and R2 have no value; only the evaluation of undefined scalar objects is erroneous (RM 3.2.1/18).

Changes from July 1982

s7. There are no significant changes.

Changes from July 1980

s8. The value of 'CONSTRAINED' for a formal parameter now depends on the mode of the formal parameter and on whether the formal parameter is constrained.

Legality Rules

L1. The assignment, aggregate formation, equality, and inequality operations are not defined for limited record types (RM 3.7.4/1, 5).

- L2. The operation of type conversion must only be applied between a parent and derived type or between two types derived from the same parent (RM 4.6/9).
- L3. The prefix for the 'CONSTRAINED attribute must denote an object (including a formal parameter or component of a composite object) that has discriminants (RM 3.7.4/3), or the prefix must denote a private type (with or without discriminants) (RM 7.4.2/9).

Test Objectives and Design Guidelines

- T1. Check that the 'CONSTRAINED attribute cannot be applied to a record, private, or limited private object whose type has no discriminants, or to an array object, or to the value of a function returning a record object that has discriminants.

Implementation Guideline: Check both for objects designated by an access value and for objects declared directly or as formal parameters of subprograms and generic units.

Check that 'CONSTRAINED cannot be applied to a record type with discriminants.

Check that 'CONSTRAINED cannot be applied to a function returning an access value designating a record with discriminants nor to an access variable designating such an object.

- T2. Check that when a formal parameter of a subprogram, entry, or a generic unit has an unconstrained type with discriminants that have defaults, 'CONSTRAINED is TRUE when applied to formal parameters of mode in and has the value of the actual parameter for the other modes.
- T3. Check that when a formal parameter of a subprogram, entry, or a generic unit has an unconstrained type with discriminants that do not have defaults, 'CONSTRAINED is TRUE when applied to formal parameters of any mode.

- T4. Check that 'CONSTRAINED is TRUE for variables declared with a constrained type, for constant objects (even if not declared with a constrained type), and designated objects.

Implementation Guideline: Use types with discriminants that do and do not have defaults.

Check that 'CONSTRAINED is FALSE for unconstrained variables that have discriminants with default values.

- T5. Check that when assigning to a constrained or an unconstrained object of a type declared with default discriminants, the assignment does not change the 'CONSTRAINED value of the object assigned to.

Implementation Guideline: The assignments should use values whose CONSTRAINED attribute's value differs from that of the object being assigned to.

- T6. Check that the operations of assignment and aggregates are defined for nonlimited record types (implicitly checked by other tests) and not defined for limited record types (see IG 7.4.4/T4 for assignment and IG 4.3/T1 for aggregates).
- T7. Check that membership tests (see IG 4.5.2.g/T74 and IG 4.5.2.i/T91), component selection (see IG 4.1.3/T1), and qualification (see IG 4.7/T2) are defined for limited and nonlimited record types.
- T8. Check that the operation of type conversion for record types is defined only between derived types having a common ancestor (see IG 4.6/T51).
- T9. Check that the attribute 'BASE is defined for record types and subtypes, but not for record objects.

Check that the attribute 'ADDRESS is defined for objects of record types but not for the record type itself (see IG 13.7.2/T1).

Check that 'SIZE is defined for record objects and record types (see IG 13.7.2/T3).

Check that 'POSITION, 'FIRST_BIT, and 'LAST_BIT are defined for components of record objects (see IG 13.7.2/T4).

- T10. Check that the operations of predefined equality and inequality are defined for objects of record types (implicitly checked by other tests) unless the record type is limited (see IG 7.4.4/T4).
- T11. Check that the operations of assignment, comparison, membership tests, qualification, type conversion, 'BASE, 'SIZE, and 'ADDRESS are defined for null records.

3.8 Access Types

Semantic Ramifications

S1. Values having an access type are, in essence, pointers. Unlike in some languages, access values can only designate (point to) objects having a specified type. Moreover, the objects designated by access values are disjoint from objects declared by object declarations. Hence, if A.all has type INTEGER and X is a local variable (not a formal parameter) having type INTEGER, it can never be the case that an assignment to A.all changes the value of X, and vice versa. In addition, the object pointed to can never be a component of another object, i.e., if A and B have the same access type and $A \neq B$, or if A and B have different access types and do not share the same collection, an assignment to A.all can never change the value of B.all.

S2. Of course, UNCHECKED_CONVERSION and the ADDRESS attribute can be used to obtain a pointer to a local variable or a component of a designated object, but an implementation is not required to take such uses of UNCHECKED_CONVERSION into account. It is up to the user of UNCHECKED_CONVERSION to conform to the normal rules of Ada, not vice versa (RM 13.10.2/3). Hence, optimizations can depend on the above statements about how assignments leave the value of certain other objects unchanged.

S3. From an implementation viewpoint, if an access type is declared with a constrained subtype indication, e.g.:

```
subtype SM_INT is INTEGER range 1..100;
type ACC_SM_INT is access SM_INT;
```

all variables, constants, and components of type ACC_SM_INT either have the value null or point to an integer object whose value is either undefined or in the range 1 through 100.

S4. Different access types may have different representations in two senses (RM 13.2/4, /7). First, the size of the collection associated with the type may be specified explicitly, giving a lower bound on the number of access objects that can be allocated. Second, the number of bits occupied by an access value may be made sufficiently small that, in effect, an offset pointer representation must be used. In such a case, the base address for the offset pointer could be the address of the beginning of the collection, or it might be some address in the middle of the collection, depending on the indexing capabilities of the target machine.

S5. Unless a programmer instantiates the UNCHECKED_DEALLOCATION procedure for an access type (see RM 13.10.1), there is no operation available for explicitly freeing allocated access objects. If unneeded storage is to be freed, a garbage collector must do it. However, there is no requirement to provide a garbage collector, and under some circumstances an implementation may choose to provide garbage collection only for certain access types, e.g., access types that are not shared among tasks.

S6. The null access value satisfies any subtype constraint for an access variable, e.g.:

```
JOHN : ACC_PERSON (M) := null;
MARY : ACC_PERSON (F) := null;
```

The assignment `JOHN := MARY` is legal and does not raise `CONSTRAINT_ERROR` as long as `MARY` has the value `null`.

S7. For access type declarations in which index or discriminant constraints are imposed as part of the declaration, e.g.:

```
type ACC_MALE is access PERSON (M);
type ACC_BOARD is access MATRIX (8, 8);
```

no constraints can be applied subsequently when the type name is used, e.g.:

```
JOHN : ACC_MALE(M);      -- illegal
CHESS : ACC_BOARD(8, 8); -- illegal
```

`ACC_MALE` already imposes a discriminant constraint so it is illegal to provide another constraint (RM 3.3.2/5). Similarly, no index constraint can be imposed on `ACC_BOARD`.

S8. Although range and accuracy constraints can be used in the declaration of an access type, only index and discriminant constraints are allowed after the name of an access type in a subtype indication (RM 3.8/6):

```
type ACC_I is access INTEGER range 0..100; -- ok
X : ACC_I range 10..20;                   -- illegal
```

S9. No constraint can be given for a type whose designated type is an access type, e.g.:

```
type ACC_STR is access STRING;
type ACC_ACC_STR is access ACC_STR;
X : ACC_STR (1..3);           -- legal
Y : ACC_ACC_STR (1..3);       -- illegal
```

The subtype indication in `Y`'s declaration is illegal because the designated type must be an unconstrained array type (RM 3.6.1/3); the designated type cannot be an access type. A similar rule holds for discriminant constraints (RM 3.7.2/1).

S10. An unconstrained array type must be used with an index constraint in an object or component declaration (RM 3.6.2/6), but a constraint need not be given in an access type declaration:

```
X : STRING;                -- illegal
type ACC_STR is access STRING; -- ok
Y : ACC_STR;                -- ok: no constraint required
```

When a designated object is created for `Y`, the allocator will provide the necessary index constraint (RM 3.6.1/8). Similar observations apply for types with discriminants.

S11. When an index constraint is given for an access type, the compatibility of the index values must be checked using the normal definition of compatibility for an index constraint (RM 3.6.1/4). When a discriminant constraint is given for an access type, only the compatibility of the discriminant values with the discriminant subtypes must be checked (AI-00007; see also IG 3.7.2/S), e.g.:

```
subtype SM is INTEGER range -10..10;
type R (D : SM) is
  record
```

```

      C : STRING (D..3);
    end record;
  type ACC_R is access R;
  X1 : ACC_R (11);           -- CONSTRAINT_ERROR
  X2 : ACC_R (-1);           -- optional CONSTRAINT_ERROR

```

CONSTRAINT_ERROR is raised for X1 because 11 does not belong to the range -10..10. CONSTRAINT_ERROR is only raised for X2 if the discriminant value, -1, is checked for compatibility with its use in the designated subtype. Since -1..3 is not a compatible index constraint for STRING, CONSTRAINT_ERROR can be raised. However, AI-00007 says that this subcomponent check need not be performed. (See IG 3.7.2/S for further discussion.)

S12. When an access subtype indication is used in an access type definition, some care is required in interpreting uses of the resulting type. Consider the following examples, in which ACC_PERSON and ACC_PERSON(M) are access subtype indications:

```

  type PERSON (SEX : GENDER) is record ... end record;
  type ACC_PERSON is access PERSON;
  subtype ACC_MALE is ACC_PERSON(M);
  type ACC_ACC_PERSON is access ACC_PERSON;
  type ACC_ACC_MALE is access ACC_PERSON(M);

  PNP : ACC_ACC_PERSON;
  MNP : ACC_ACC_MALE;

```

Now suppose we execute:

```
PNP := new ACC_PERSON (new PERSON(M));
```

This allocates a pointer, PNP.all, that points to an object whose SEX component is M, i.e., PNP.all.all.SEX = M. However, the assignments:

```

PNP := new ACC_PERSON;
MNP := new ACC_PERSON;

```

allocate pointers whose value is null, i.e., PNP.all = null = MNP.all and hence the component PNP.all.all.SEX does not exist. Since constraints are not associated with allocated access values, but with allocated objects, the following sequence of assignments is legal:

```

PNP := new ACC_PERSON;
PNP.all := new PERSON(F);
PNP.all := new PERSON(M);

```

The F and M constraint values are associated with PNP.all.all.SEX.

S13. But now consider:

```

MNP := new ACC_PERSON;           -- ok
MNP.all := new PERSON(F);        -- CONSTRAINT_ERROR
MNP := new ACC_PERSON(new PERSON(F)); -- CONSTRAINT_ERROR

```

The first assignment raises no exception since the object designated by new ACC_PERSON has the value null, and this value satisfies any constraint. The allocator in the second assignment statement raises CONSTRAINT_ERROR since MNP's designated subtype is ACC_PERSON(M), and the access value created by new PERSON(F) does not belong to ACC_PERSON(M); it is not null and it does not designate a PERSON object whose discriminant has the value M.

In the third assignment, the outermost allocator must return a value of type ACC_ACC_MALE (since this value is being assigned to MNP), and hence the inner allocator must return a value belonging to ACC_PERSON(M). Since the inner allocator does not do so, CONSTRAINT_ERROR is raised.

Approved Interpretations

S14. When a discriminant constraint is given for an access type, an implementation need only check that the discriminant values are compatible with the discriminant subtypes of the designated type (AI-00007).

S15. A non-null access value of type T belongs to every subtype of T if the designated type is neither an array type nor a type with discriminants (AI-00324).

Changes from July 1982

S16. There are no significant changes.

Changes from July 1980

S17. There are no significant changes.

Legality Rules

- L1. The only forms of constraint allowed after the name of an access type in a subtype indication are index constraints and discriminant constraints (RM 3.8/6).
- L2. If an index constraint is given in a subtype indication and the type mark denotes an access type,
 - the access type must not already be constrained (RM 3.3.2/5),
 - the designated type must be an unconstrained array type (RM 3.6.1/3),
 - one discrete range must be provided for every index of the array type; and
 - the base type of each discrete range must be the same as the base type of the corresponding array index.
- L3. If a discriminant constraint is given in a subtype indication and the type mark denotes an access type,
 - the access type must not already be constrained (RM 3.3.2/5),
 - the designated type must be an unconstrained type with discriminants (RM 3.7.2/1) (including an unconstrained incomplete type if the subtype indication occurs in an access type definition; RM 3.8.1/4),
 - if the constraint contains both positional and named associations, the positional associations must be given first (RM 3.7.2/3);
 - a discriminant name in the constraint must be the name of a discriminant of the type being constrained (RM 3.7.2/4);
 - a discriminant association with more than one discriminant name is only allowed for discriminants having the same type (RM 3.7.2/4);
 - the base type of a discriminant specified in the discriminant constraint and the corresponding discriminant in the designated type must be the same (RM 3.7.2/4); and

- the constraint must specify exactly one value for each discriminant (RM 3.7.2/4).

Exception Conditions

- E1. If an index constraint is given for an access type in a subtype indication, `CONSTRAINT_ERROR` is raised if either bound of a non-null discrete range of the constraint does not belong to the corresponding index subtype of the designated type (RM 3.6.1/4, RM 3.5/4, and RM 3.3.2/9).
- E2. If a discriminant constraint is given for an access type in a subtype indication, `CONSTRAINT_ERROR` is raised if the value specified for any discriminant does not lie in the permitted range of values for the discriminant (RM 3.3.2/9 and RM 3.7.2/5).
- E3. If the type mark in a subtype indication denotes an access type and the subtype indication contains a discriminant constraint, `CONSTRAINT_ERROR` may, but need not be raised when the subtype indication is elaborated, if a specified discriminant value is incompatible with its use in a component subtype definition of the designated type (AI-00007 and RM 3.7.2/5). In addition, `CONSTRAINT_ERROR` may, but need not be raised if some other discriminant value in such a component subtype definition is not compatible with the component type (RM 3.7.2/5 and AI-00007).

Test Objectives and Design Guidelines

- T1. Check that record, array, and access type definitions are forbidden in access type definitions (e.g., forms like `access record ... end record` are forbidden).
- T2. Check that an unconstrained array type or a record type without default discriminants can be used in an access type definition without an index or discriminant constraint. Check that (nonstatic) index or discriminant constraints can subsequently be imposed when the type is used in an object declaration, array component declaration, record component declaration, access type declaration, parameter declaration, allocator (see IG 4.8/T5), and derived type definition.
Implementation Guideline: Include a check when the unconstrained type is a generic formal type and when the access type is a generic formal access type.
- T3. Check that if an index or discriminant constraint is provided in an access type definition (or if the subtype indication is already constrained), the access type name cannot subsequently be used with an index or discriminant constraint in an object declaration, derived type declaration, array component declaration, record component declaration, access type declaration, parameter declaration, allocator (see IG 4.8/T2), and derived type definition.
Implementation Guideline: Include a check for generic formal access types.
- T4. Check that an index or discriminant constraint for an access type can reference a discriminant value in a record component declaration, e.g.:

```

type ACC_STRING is access STRING;
type TEXT(L : INTEGER) is
  record
    VALUE : ACC_STRING(1..L);
  end record;

```

and that slicing and indexing can be applied to such a component.

Implementation Guideline: Include a case where the access type is declared as a generic formal type.

- T5. Check that all access objects, including array and record components, are initialized by default with the value null.

Implementation Guideline: Include a case where the object is declared with a formal private type and the actual type in an instantiation is an access type.

- T6. Check that the object accessed by a constant access object can be modified.
- T8. Check that an access type used in a subtype indication cannot be constrained with a range constraint or an accuracy constraint.

Check that a constrained access subtype (i.e., a subtype with an index or discriminant constraint specified) cannot be further constrained in a subtype indication, even if the same constraint values are used.

Implementation Guideline: This objective differs from T3 because it refers to access subtype names introduced by a subtype declaration, whereas T3 refers to constraints specified in an access type definition.

Implementation Guideline: Try the above checks for access types whose object is an access type as well as for access types whose object is a nonaccess type.

- T9. Check that an index or discriminant constraint cannot be imposed on an access type whose designated type is an access type.

Implementation Guideline: The designated type should itself designate a suitable unconstrained array type or type with discriminants.

Implementation Guideline: Include checks for generic formal access types.

- T10. Check that an index constraint must have the correct number of dimensions and index types when imposed on an access type (see IG 3.6.1.b/T71).
- T11. Check that a discriminant constraint must be correctly formed when imposed on an access type (see IG 3.7.2/T1).
- T12. Check that CONSTRAINT_ERROR is raised when an incompatible index constraint is imposed on an access type (see IG 3.6.1.b/T72).
- T13. Check that CONSTRAINT_ERROR is raised when a discriminant constraint is imposed on an access type and a discriminant value does not belong to the discriminant's subtype (see IG 3.7.2/T11).

3.8.1 Incomplete Type Declarations

Semantic Ramifications

S1. The type mark declared in an incomplete type declaration must be completely declared later in either a full type declaration or a task type declaration. If the incomplete declaration appears in the visible part of a package specification, the full declaration must appear later in the same visible part (i.e., not in the private part) and not in a visible part of an enclosed package, e.g.;

```
package P is
  type T;
  type U;
  package Q is
    type T is access INTEGER;    -- does not complete P.T
  end Q;
private
  type U is access INTEGER;    -- illegal; not in visible part
end P;
```

S2. If the incomplete declaration appears in a private part, the complete declaration must appear later in the same private part or in the package body. If the complete declaration does not appear in the private part, then a package body is required.

S3. The full declaration of an incomplete type cannot be a private type declaration:

```
package P is
  type T;
  type T is private;      -- illegal
end P;
```

The private type declaration does not complete T's declaration because RM 3.8.1/3 requires either a task type declaration or a full type declaration, and a "full type declaration" is a syntactic form that excludes private type declarations (RM 3.3.1/2). Similarly, an incomplete type declaration is not allowed as the full declaration of a private type.

S4. The rules applicable to the full declaration of an incomplete type are not the same as those for the full declaration of a private type. In particular, if the incomplete type does not have discriminants, the full type can be an unconstrained array type or a derived type that has discriminants:

```
package P is
  type T;
  type U;
  type VSTR (SIZE : POSITIVE) is
    record
      DATA : STRING (1..SIZE);
    end record;
  ...
  type T is new STRING;    -- legal
  type U is new VSTR;      -- legal
end P;
```

S5. An incomplete type cannot be used in its own full declaration:

```
type R;                      -- (1)
type AR is access R;
type R is                    -- (2)
  record
    A : AR := new R;        -- illegal (3)
  end record;
```

The use of R at (3) is illegal because R's full declaration is not yet visible (RM 8.3/5) and the incomplete type declaration at (1) is hidden by the full type declaration at (2) (AI-00386). Similarly, the following full declaration is illegal:

```
type S;
type S is access S;          -- illegal
```

The third occurrence of S is illegal because the second occurrence is not yet visible and the first occurrence is hidden by the second occurrence.

S6. It is possible to define a sequence of access types that chase themselves:

```
type T;
type U;
type T is access U;
type U is access T;
```

This sequence of declarations is legal. The full declaration of T refers to U's incomplete declaration, and the full declaration of U refers to T's full declaration.

S7. If an incomplete type declaration contains a discriminant specification, the full declaration must declare a record type, since the full declaration must also contain a discriminant specification (RM 3.8.1/4), and a full declaration with a discriminant specification must declare a record type (RM 3.7.1/3).

S8. Additional operations for an access type may need to be declared after the full declaration of an incomplete type (see RM 7.4.2/8). For example:

```
package P is
  type T;
  type ACC_T is access T;
  A : ACC_T;
  type T is new STRING;      -- indexing etc. declared for ACC_T
end P;

J : INTEGER := P.A'FIRST;    -- legal
```

When T's full declaration is given, it is known that ACC_T's designated type is an array type, so the appropriate operations for such an access type must be declared. These are declared at the earliest place within the immediate scope of the access type and after the full declaration for T (RM 7.4.2/8 and RM 7.4.2/7). In this case, the earliest place is immediately after T's full declaration. The place where these declarations occur affects the legality of operations applied to ACC_T or the variable A. In particular, A'FIRST is illegal prior to T's full declaration and legal afterwards. The declaration of the attribute FIRST for ACC_T in P's visible part means this operation is visible outside P for P.A.

S9. More complex cases of deferred declaration of operations can occur. See IG 7.4.2/S for further discussion.

S10. When a discriminant constraint is given for an incomplete type (or for an incompletely declared private type, or an access type that designates an incompletely declared type), RM 3.7.2/5 requires that the discriminant values be checked for compatibility with their use within the complete type declaration. For an incomplete type declared in the private part of a package, the complete declaration could be given in a separately compiled package body, so it is not possible to perform the required compatibility check when the discriminant constraint is elaborated. When the complete declaration occurs later in the same compilation unit, it is possible to defer the check until the complete declaration is elaborated. The full declaration of an incomplete or access type may still contain references to an incompletely declared type, thereby requiring further deferral of the required compatibility check:

```
package P is
  subtype INT7 is INTEGER range 1..7;
  subtype INT6 is INTEGER range 1..6;

  type T_INT7 (D7 : INT7);
  type T_INT6 (D6 : INT6);

  type ACC_T_INT7 is access T_INT7;
  type ACC_T_INT6 is access T_INT6;

  subtype ACC_T_CONS is ACC_T_INT7(6);
```



```

type T_INT7 (D7 : INT7) is           -- full declaration
  record
    C76 : ACC T_INT6(D7);           -- but not complete
  end record;

type T_INT6 (D6 : INT6) is
  record
    CF : STRING (FUNC .. D6);
  end record;
end P;
```

The full declaration of T_INT6 completes the declaration of T_INT7 and allows the compatibility check required for ACC_T_CONS to be completed.

Approved Interpretations

S11. A discriminant constraint cannot be given for an incomplete type declared in the private part of a package if the full declaration of the type is given in the package body (AI-00007).

Changes from July 1982

S12. The full declaration of an incomplete type can be a task type.

S13. The full declaration must appear immediately within the same visible part or declarative part as the incomplete type declaration.

S14. If the incomplete declaration is given in a private part, the full declaration can appear in the body.

Changes from July 1980

S15. Only a discriminant constraint can be given for an incomplete type.

S16. The full declaration of an incomplete type can declare an unconstrained array type or a derived type with discriminants.

Legality Rules

- L1. If an incomplete type declaration appears in the visible part of a package, the corresponding complete declaration must appear later in the same visible part, but not within any nested package specifications (RM 3.8.1/3).
- L2. If an incomplete type declaration appears in the declarative part of a block, subprogram body, package body, or task body, the corresponding complete declaration must appear later in the same declarative part, but not within any nested declarative parts or package specifications (RM 3.8.1/3).
- L3. If an incomplete type declaration appears in a private part of a package, the corresponding complete declaration must appear either later in the same private part (but not within any nested package specifications) or immediately within the declarative part of the corresponding package body (RM 3.8.1/3).
- L4. If an incomplete type declaration contains a discriminant part, the corresponding full declaration must have a conforming discriminant part (see IG 6.3.1/L) and must be a record type definition (RM 3.8.1/4 and RM 3.7.1/3).
- L5. If an incomplete type declaration does not contain a discriminant part, the full type declaration must not contain a discriminant part (RM 3.8.1/4).
- L6. The declaration that completes the declaration of an incomplete type must be either a full

type declaration or a task type declaration (RM 3.8.1/3). (It cannot be a private type declaration.)

- L7. Prior to the end of the full declaration for an incomplete type, the only allowed use of a name that denotes the incomplete type is as the type mark in the subtype indication of an access type definition; the only form of constraint allowed in this subtype indication is a discriminant constraint (RM 3.8.1/4), and a discriminant constraint is only allowed if 1) the incomplete type was declared with a discriminant part (RM 3.7.2/1), and 2) the incomplete type declaration and its corresponding full declaration occur in the same declarative part, visible part, or private part (AI-00007).
- L8. If a discriminant constraint is given in a subtype indication and the type mark is an incomplete type (this can only occur in an access type definition; see L7):
- the incomplete type must have been declared with a discriminant specification (RM 3.7.2/1);
 - if the constraint contains both positional and named associations, the positional associations must be given first (RM 3.7.2/3);
 - a discriminant name in the constraint must be the name of a discriminant of the incomplete type (RM 3.7.2/4);
 - a discriminant association with more than one discriminant name is only allowed for discriminants having the same type (RM 3.7.2/4);
 - the base type of a discriminant specified in the discriminant constraint and the corresponding discriminant of the incomplete type must be the same (RM 3.7.2/4); and
 - the constraint must specify exactly one value for each discriminant (RM 3.7.2/4).

Exception Conditions

- E1. If a discriminant constraint is given for an incomplete type, `CONSTRAINT_ERROR` is raised if the value specified for any discriminant does not lie in the permitted range of values for the discriminant (RM 3.3.2/9 and RM 3.7.2/5).
- E2. If the full declaration for an incomplete type with discriminants, `T`, contains a component subtype definition that is dependent on a discriminant (i.e., if the component subtype definition contains an index or a discriminant constraint that uses a discriminant of the enclosing type) and a subtype indication giving a discriminant constraint for `T` is elaborated before `T`'s complete declaration, `CONSTRAINT_ERROR` is raised no later than the end of `T`'s complete declaration if a specified discriminant value is not compatible with its use in specifying the component's subtype (RM 3.7.2/5 and AI-00007), i.e., `CONSTRAINT_ERROR` is raised if:
- the discriminant is used in a discrete range of an index constraint, the range is not null, and at least one bound does not belong to the index subtype; or
 - the discriminant is used in a discriminant constraint and the value of any discriminant in the constraint does not lie in the range of values permitted for the discriminant.

In addition, `CONSTRAINT_ERROR` is raised no later than the end of `T`'s complete declaration if some other discriminant value in such a component subtype definition is not compatible with the component type (RM 3.7.2/5 and AI-00007).

Test Objectives and Design Guidelines

- T1. Check that if an incomplete type declaration appears in the visible part of a package, the full declaration must appear in the same part and in particular, 1) cannot be omitted; 2) cannot be given in the private part; 3) cannot appear in the package body's declarative part; 4) cannot appear in a package specification nested in the visible or private part containing the incomplete declaration.

Check that if an incomplete type declaration appears in the private part of a package, the full declaration need not appear immediately within the same part, and if it does not appear in the private part, a package body containing the complete declaration is required (see IG 7.3/T1).

Implementation Guideline: Include tests for generic package declarations and when the body is separately compiled or is given as a subunit. See also T8.

Check that if an incomplete type declaration appears in the declarative part of a block, subprogram body, package body, or task body, the corresponding complete declaration must appear in the same declarative part, excluding any nested declarative parts or package specifications.

Check that the full declaration of an incomplete type cannot be a private type declaration.

Implementation Guideline: Declare an incomplete type in the visible part of a package.

- T2. Check that an incomplete type declaration can be given for any type, i.e., that the corresponding full declaration can declare an integer type, a real type, an enumeration type, a constrained or unconstrained array type, a record type without discriminants (types with discriminants are checked below), an access type, a task, or a derived type (including a derived type that is an unconstrained array type or an unconstrained type with discriminants).
- T3. Check that if an incomplete type is declared with discriminants, the complete declaration must have a conforming discriminant part and cannot be a derived type with discriminants.
- Implementation Guideline:* Include separately compiled package bodies and subunits.
- T5. Check that prior to its complete declaration, an incompletely declared type:

- cannot be used in an object declaration, subtype declaration, component declaration, or parameter declaration, or as the component type in an array declaration;
- cannot be used with a range constraint, accuracy constraint, or index constraint in an access type definition, even if the full declaration would permit such constraints;
- cannot be used with a discriminant constraint if none was present in the incomplete type declaration.
- cannot be used as an actual parameter in a generic instantiation.

Check that an access type declared with an incomplete type cannot be used with an index constraint or discriminant constraint until after the full declaration has been elaborated.

- T6. Check that additional operations are declared for an access type when the incomplete type is fully declared (see IG 7.4.2/T5).
- T7. For an incomplete type with discriminants declared in the visible part of a package or in a declarative part, check that `CONSTRAINT_ERROR` is raised if a discriminant constraint is specified for the type and one of the discriminant values does not belong to the corresponding discriminant's subtype.

Implementation Guideline: Repeat the above check for an incomplete type declared in a private part when the full declaration is also given in the private part.

Similarly, if a discriminant constraint is applied to an access type that designates an incomplete type declared in the visible or private part of a package or in a declarative part, check that `CONSTRAINT_ERROR` is raised if one of the discriminant values does not belong to the corresponding discriminant's subtype.

Implementation Guideline: Include a case where the full declaration is in the package body.

When an incomplete type with discriminants is declared in a private part and its full declaration is not also given in the private part, check that a discriminant constraint cannot be applied to the incomplete type.

Check whether a deferred check is performed when a discriminant constraint is given for an access type that designates an incomplete type (see IG 3.7.2/T17).

Check that a deferred check is performed no later than the end of an incomplete type's complete declaration if a discriminant constraint is applied to an incomplete type before its complete declaration (see IG 3.7.2/T14, /T16).

- T8. Check that when an incomplete type declaration is given in the private part of a package, the full type declaration can appear in the package body.

Implementation Guideline: Check that the body can be separately compiled or a subunit.

3.8.2 Operations of Access Types

Semantic Ramifications

- S1. The operations declared for an access type are:

basic operations (RM 3.8.2/1)

assignment

allocator

membership tests

qualification

conversion

the literal null

selectors for discriminants of the designated type, if any

selectors for components of the designated type if it is a

record type

formation of indexed components, if the designated type is an

array type

formation of slices, if the designated type is a one-dimensional

array type

selection of entries and entry families if the designated type

is a task type

selected component with selector all

operators (RM 3.8.2/5)

equality and inequality

attributes

ADDRESS (RM 13.7.2/3)

BASE (RM 3.3.3/9)

SIZE (RM 13.7.2/4)

STORAGE_SIZE (RM 13.7.2/12)

if the designated type is an array type:

FIRST (RM 3.8.2/2)

LAST (RM 3.8.2/2)

RANGE (RM 3.8.2/2)

LENGTH (RM 3.8.2/2)

if the designated type is a task type:

TERMINATED (RM 3.8.2/3)

CALLABLE (RM 3.8.2/3)

S2. If a designated type is an incomplete type, and the full declaration declares either a record type, a type with discriminants (when the incomplete type had none), or a task type, additional operations are declared for the access type at the earliest place within the immediate scope of the access type (RM 7.4.2/8). These additional operations are component selection operations for the record components or discriminants, selection operations for the task entries or entry families, indexing and slicing operations, and appropriate attributes. See IG 3.8.1/S and IG 7.4.2/S for further discussion.

S3. Although the array attributes FIRST, LAST, LENGTH, and RANGE are declared when the designated type is an array type, the prefix of these attributes is not allowed to be a type mark; the prefix must have an access type (RM 3.8.2/2).

Changes from July 1982

S4. Operations for selecting entry families are mentioned in the list of operations for a designated task type.

S5. FIRST, LAST, LENGTH, and RANGE are not allowed for an access type that designates an array type.

S6. The attributes CALLABLE and TERMINATED are defined for values of an access type that designates a task type.

S7. The SIZE attribute is defined for an access type.

Changes from July 1980

S8. The prefix of FIRST, LAST, LENGTH, and RANGE can be a value of an access type if the designated type is an array type.

Legality Rules

- L1. If the prefix of FIRST, LAST, LENGTH, RANGE, FIRST(N), LAST(N), LENGTH(N), and RANGE(N) denotes an object or value having an access type, the designated type must be an array type with N or fewer dimensions and N must be a static *universal_integer* expression with a value greater than zero. (RM 3.8.2/2 and RM 3.6.2/2).
- L2. If the prefix of TERMINATED or CALLABLE denotes an object or value having an access type, the designated type must be a task type (RM 3.8.2/3).
- L3. If the prefix of an indexed component denotes an object or value having an access type, the designated type must be an array type with the correct number of dimensions and index types (RM 4.1.1/3).
- L4. If the prefix of a slice denotes an object or value having an access type, the designated type must be a one-dimensional array type, and the index type of the slice must be the same as the index type of the array (RM 4.1.2/3).
- L5. If the prefix of a selected component denotes an object or value having an access type, then the designated type must be either a record type or a type with discriminants and the

selector must be the name of a component or discriminant (respectively), or the designated type must be a task type, and the selector must be the name of an entry or entry family (RM 4.1.3/3-10).

- L6. If the prefix of an indexed component, selected component, slice, or attribute has an access type, then the prefix must not be a name that denotes a formal parameter of mode out or a subcomponent thereof (RM 4.1/4).

Exception Conditions

- E1. `CONSTRAINT_ERROR` is raised if the prefix of the attribute `FIRST`, `LAST`, `LENGTH`, or `RANGE` has the access value, null (RM 4.1/10).

Test Objectives and Design Guidelines

- T1. Check that `FIRST`, `LAST`, `LENGTH`, `RANGE`, `FIRST(N)`, `LAST(N)`, `LENGTH(N)`, and `RANGE(N)` are declared, and return the correct values, for objects and the result of function calls having an access type whose designated type is an array type (see IG 3.6.2/T4).

Implementation Guideline: Include checks of generic formal array parameters and array types whose indexes are given by formal scalar types.

- T2. Check that `TERMINATED` and `CALLABLE` are declared and return the correct values for objects and the result of function calls having an access type whose designated type is a task type.
- T3. Check that the prefix of `FIRST`, `LAST`, `LENGTH`, `RANGE`, `FIRST(N)`, `LAST(N)`, `LENGTH(N)`, and `RANGE(N)` cannot be an access type that designates a constrained array type.
- T4. Check that the prefix of `TERMINATED` and `CALLABLE` cannot be an access type that designates a task type.

3.9 Declarative Parts

Semantic Ramifications

S1. Pragmas may precede or follow any declaration in a declarative part (RM 2.8/4-5). In addition, a declarative part may consist solely of pragmas. Only certain pragmas are defined to have an effect if they appear in a declarative part (and are otherwise used correctly). These are: `CONTROLLED` (RM 4.8/11), `INLINE` (RM 6.3.2/2-3), `INTERFACE` (RM 13.9/2-3), `LIST` (RM B/6), `OPTIMIZE` (RM B/8), `PACK` (RM 13.1/11-12), `PAGE` (RM B/10), `PRIORITY` (in declarative parts of library subprograms; RM 9.8/2), `SHARED` (RM 9.11/10), and `SUPPRESS` (RM 11.7/3).

S2. The following forms of declaration are not allowed after a body is declared in a declarative part: an object declaration, a type declaration (but not the declaration of a task type), an exception declaration, a renaming declaration, a number declaration, or a subtype declaration. In addition, representation clauses are not allowed after the occurrence of a body. However, subprogram declarations, single task declarations, task type declarations, use clauses, package declarations, generic declarations, and generic instantiations are allowed both before and after the occurrence of a body.

S3. Although certain forms of declaration are not allowed directly after a body declaration, it is always possible to include such declarations in a package specification, e.g.:

```

function F return INTEGER is ... end F;

X : INTEGER := F;      -- illegal; occurs after a body is declared

package DUMMY is
  X : INTEGER := F;    -- legal; a package declaration is allowed
end DUMMY;
use DUMMY;

```

The effect is almost the same as allowing an object declaration directly.

S4. The syntax allows an empty declarative part, e.g.:

```

declare
begin      -- no declarations required
  ...
end;

```

An otherwise empty declarative part can, of course, contain pragmas.

S5. A subprogram body may appear without a corresponding subprogram declaration (RM 6.3/3). If both appear, then RM 3.9/9 requires that the declaration appear first. If an INTERFACE pragma is specified for the subprogram, a body is not allowed (RM 13.9/3).

S6. An attempt to activate a task before elaboration of its body occurs most straightforwardly for access types that designate task types:

```

task type T;
type ACC_T is access T;
V : ACC_T;
W : ACC_T := new T;      -- activation attempt raises PROGRAM_ERROR
task body T is ... end T;

```

S7. A premature attempt to activate a task can also occur without using access types:

```

task type T;

package P is
  R : T;      -- legal object declaration
end P;

package body P is
end;      -- activation attempt raises PROGRAM_ERROR (RM 9.3/5, RM 9.3/2)

task body T is ... end;

```

S8. The requirement to elaborate a task's body before the task is activated does not preclude calling an entry before the task is active:

```

package P is end;

task T is
  entry E;
end T;

package body P is
  task PT;

```

```

    task body PT is
        T.E;
    end PT;

begin
    -- PT is activated when the package body is elaborated (RM 9.3/2)
    -- so T can be called, even though it is not yet activated; no
    -- exception is raised even though T's body has not been
    -- elaborated.
    null;
end P;

task body T is ... end;
```

S9. If a set of tasks is to be activated, the elaboration checks must be performed for all task bodies before any tasks are activated (AI-00149). For example, suppose activation of three tasks is required and each task has a different body. Suppose the activation attempt for one task would raise PROGRAM_ERROR because the task's body has not yet been elaborated. Suppose an attempt to activate another task would raise TASKING_ERROR because an exception will occur while elaborating the task's declarative part (RM 9.3/3). Finally, assume that an attempt to activate one of the tasks would succeed:

```

task type NORMAL_ACTIVATION;

task type RAISE_PROGRAM_ERROR;

task type RAISE_TASKING_ERROR;

type REC is
    record
        A : NORMAL_ACTIVATION;
        B : RAISE_PROGRAM_ERROR;
        C : RAISE_TASKING_ERROR;
    end record;

task body NORMAL_ACTIVATION is
begin
    null;
end NORMAL_ACTIVATION;

task body RAISE_TASKING_ERROR is
    X : INTEGER := 1/0;
begin
    ...
end RAISE_TASKING_ERROR;

package ACTIVATE_TASKS is
    X : REC;
end ACTIVATE_TASKS;

package body ACTIVATE_TASKS is
end ACTIVATE_TASKS;      -- tasks in X are to be activated here

task body RAISE_PROGRAM_ERROR is ... end;
```


The RM allows the components of `ACTIVATE_TASKS.X` to be activated in parallel (RM 9.3/1), but before any activation is started, an elaboration check must be performed (AI-00149). Hence, if `PROGRAM_ERROR` is raised, no tasks have been activated.

S10. Elaboration occurs at run-time (RM 3.1/8). This means that calls to subprograms can textually appear before the subprogram body without causing `PROGRAM_ERROR` to be raised:

```
function F return INTEGER;

procedure PC is
  X : INTEGER := F;  -- no PROGRAM_ERROR when PC is called
begin
  ...
end PC;

function F return INTEGER is ... end;
```

The call to `F` does not raise `PROGRAM_ERROR` because `PC` is not invoked until after `F`'s body has been elaborated.

S11. The possibility of calling a subprogram before its body is elaborated cannot always be detected at compile time. For example,

```
package P is ... end;

function F return INTEGER;

procedure PROC is
  I : INTEGER;
begin
  if ... then
    I := F;
  end if;
end PROC;

package body P is
begin
  PROC;
end P;

function F return INTEGER is ... end;
```

Whether or not `F` is called depends on the value of the condition in `PROC`'s `if` statement. Moreover, if `PROC` is called after `F`'s body has been elaborated, the call to `F` must not raise `PROGRAM_ERROR`, so the call to `F` in `PROC` cannot simply be replaced with `raise PROGRAM_ERROR`. Similar cases can be constructed for calls that depend on the value of a case expression or on the raising of an exception, etc.

S12. Since the invocation of a subprogram prior to elaboration of its body cannot always be detected at compile time, an implementation must be prepared to perform a run-time check. This check is easy to do if calls are made indirectly through a pointer. Initially the pointer should access code that will raise `PROGRAM_ERROR` if the call is executed. When the body is elaborated, the pointer should be updated to point to the actual code. (Similar arguments imply that elaboration checks for tasks and generic units must, in general, be performed at run-time.)

Although the possibility of calling an unelaborated subprogram body cannot, in general, be ruled

out by compile-time program analysis, in many if not most actual programs a simple compile-time analysis will suffice to show that a subprogram body will be elaborated before it can be called, and so make a run-time check unnecessary. Similar simple analyses will often suffice to eliminate run-time elaboration checks for tasks and generic units.

S13. If an INTERFACE pragma has been specified for a subprogram, no body can be provided (RM 13.9/3). The lack of a body does not mean that PROGRAM_ERROR must be raised for all calls to such a subprogram; instead, an implementation is allowed (but not required) to perform such a check (see AI-00180). The implementation must specify (in Appendix F) whether or not such a check is made, and if made, when the body is considered to be elaborated.

S14. The instantiation of a generic unit with a generic subprogram parameter can cause PROGRAM_ERROR to be raised:

```
function F return INTEGER;

generic
  with function FF return INTEGER;
package P is
  Y : INTEGER;
end P;

package body P is
begin
  Y := FF;
end P;

package N is
  package NEW_P is new P(F);  -- PROGRAM_ERROR
end N;

function F return INTEGER is ... end;
```

PROGRAM_ERROR is raised because NEW_P's body is elaborated when the instantiation is elaborated, and the elaboration of NEW_P's body requires that function F be called before F's body has been elaborated.

S15. A package body is optional under certain circumstances (see IG 7.3/S). If a generic package is declared and an optional body is provided, PROGRAM_ERROR must be raised if the body has not yet been elaborated and an instantiation is elaborated:

```
generic
package P is
  X : INTEGER;
end P;

package NEW_P is new P;  -- PROGRAM_ERROR because of later body

package body P is
begin
  X := 1;
end P;
```

If no body is provided for P, the instantiation does not raise PROGRAM_ERROR. Of course, it is not always possible to detect immediately whether a body will be provided:

```

package Q is
  generic
    package P is
      X : INTEGER;
    end P;

    package N is new P;      -- possible PROGRAM_ERROR
  end P;

  -- Q's body is optional and could be compiled separately
  package body Q is
    package body P is
      begin
        X := 1;
      end P;
    end Q;

```

In this example, the existence of a package body means that the instantiation in Q's specification will raise PROGRAM_ERROR, since P's body has not yet been elaborated. If no body is provided for P (or Q!), then the instantiation must not raise PROGRAM_ERROR. Of course, if P's specification required a body, it would be easier to see at the point of the instantiation that P's body cannot yet have been elaborated.

S16. When a task object is declared in a package specification, an implicit body is provided for a package (RM 9.3/5), even when the package is created as a result of a generic instantiation. The elaboration of the implicit body requires that the task object be activated; this attempt to activate the task can raise PROGRAM_ERROR:

```

task type TT:

  generic
    type LP is limited private;
  package GP is
    X : LP;
  end GP;                      -- no body is required

  package P is new GP (TT);    -- PROGRAM_ERROR raised

  task body TT is ... end TT;

```

The attempt to activate P.X when GP is instantiated causes PROGRAM_ERROR to be raised.

S17. The order of elaboration of bodies of library units is not fully defined. Such bodies must be elaborated after their library unit declarations are elaborated and after any units named in their context clauses. The bodies must be elaborated before execution of the main program begins (RM 10.5/1-2). These rules do not ensure that library unit bodies are elaborated before they are needed by the elaboration of some other library unit. For example:

```

with SEQUENTIAL_IO;
package NEW_IO is new SEQUENTIAL_IO (INTEGER);  -- PROGRAM_ERROR?

```

Even though SEQUENTIAL_IO is a predefined library package, its body need not be considered elaborated before the NEW_IO instantiation is elaborated. If the body has not been elaborated, the instantiation will raise PROGRAM_ERROR. (The pragma ELABORATE should be used in such cases to ensure that bodies have been elaborated before they are needed.)

S18. Elaboration order checks are not performed just when declarative parts are elaborated; they are also performed when elaborating a package specification:

```
package P is
  function F return BOOLEAN;
  X : BOOLEAN := F;           -- PROGRAM_ERROR must be raised
end P;
```

S19. Declarations can also appear in task specifications but none of the declarative forms allowed in task specifications (entry declarations, pragmas, and address clauses) present the possibility of raising PROGRAM_ERROR when elaborated.

S20. Specific semantic ramifications, restrictions, and exceptions pertaining to particular constituents of a declarative part are discussed in connection with declarations (Chapter 3), representation clauses (Chapter 13), visibility (Chapter 8), subprograms (Chapter 6), packages (Chapter 7), tasks (Chapter 9), exceptions (Chapter 11), and pragmas (Section 2.8).

Changes from July 1982

S21. For generic units, an elaboration check is only performed if the generic unit has a body.

Changes from July 1980

S22. Representation clauses may appear immediately after any basic declaration; they need not be grouped together.

S23. A use clause is allowed after a package body, and in particular, after a generic instantiation.

S24. A generic unit can be declared or instantiated after a body has been declared.

S25. A subprogram declaration can be given after a body.

S26. An exception is raised (PROGRAM_ERROR) if an attempt is made to access a body before it has been elaborated.

Legality Rules

- L1. If a body is provided for a package declared in a declarative part, the body must occur after the package specification (RM 3.9/9).
- L2. If a declaration is given for a subprogram in a declarative part and no INTERFACE pragma is given for the subprogram, the corresponding body must appear in the same declarative part and cannot precede the declaration (RM 3.9/9).
- L3. The body of a task or generic unit cannot precede its declaration in a declarative part (RM 3.9/9).
- L4. If an INTERFACE pragma is given for a subprogram declared by a subprogram declaration, then no body is allowed for that subprogram (RM 13.9/3).
- L5. Two explicit declarations must not be homographs if the declarations occur immediately within the same declarative part (RM 8.3/17).
- L6. An explicit declaration that occurs immediately within the declarative part of a generic or a nongeneric subprogram body must not be a homograph of a declaration of a formal parameter of the subprogram or the generic unit (RM 8.3/17 and RM 8.1/1-7).
- L7. An explicit declaration that occurs immediately within the declarative part of a generic or a nongeneric package body must not be a homograph of a declaration that occurs immediately within the corresponding package specification or the generic formal part (RM 8.3/17 and RM 8.1/1-7).

- L8. An explicit declaration that occurs immediately within the declarative part of a task body must not be a homograph of an entry declaration that occurs in the corresponding task specification (RM 8.3/17 and RM 8.1/7).

Exception Conditions

- E1. `PROGRAM_ERROR` is raised if a subprogram's body has not been elaborated when the subprogram is called (RM 3.9/5).
- E2. `PROGRAM_ERROR` is raised if a task's body has not been elaborated when an attempt is made to activate the task (RM 3.9/5).
- E3. `PROGRAM_ERROR` is raised if a generic unit's body has not been elaborated when the unit's instantiation is elaborated (RM 3.9/5).

Test Objectives and Design Guidelines

- T1. Check that if a package declaration occurs in a declarative part, it must precede any declaration of a corresponding package body (see IG 7.3/T6).

Check that if a subprogram declaration occurs in a declarative part, it must precede the corresponding body declaration (see IG 6.3/T10 and IG 6.1/T3).

Check that if a single task declaration or task type declaration occurs in a declarative part, it must precede the corresponding body declaration (see IG 9.1/T3).

Check that the body of a generic unit must not precede its declaration (see IG 12.2/T4).

- T2. Check that two explicit declarations in the same declarative part cannot be homographs (see IG 8.3/T8).

Check that an explicit declaration that occurs within the declarative part of a generic or a nongeneric subprogram body cannot be a homograph of the declaration of a formal parameter of the subprogram or the generic unit (see IG 6.1/T4, IG 8.3/T1, IG 8.3/T4, and IG 12.1/T9).

Check that an explicit declaration that occurs within the declarative part of a generic or a nongeneric package body cannot be a homograph of a declaration that occurs in the corresponding package specification or the generic formal part (see IG 8.3/T2 and IG 8.3/T4).

Check that an explicit declaration that occurs within the declarative part of a task body cannot be a homograph of an entry declaration that occurs in the corresponding task specification (see IG 8.3/T3 and IG 9.5/T94).

- T3. Check that if an `INTERFACE` pragma is provided for a subprogram (and accepted by an implementation), no subprogram body can be provided (see IG 13.9/T1).

- T4. Check that none of the following kinds of declaration are allowed after the declaration of a subprogram body, package body, task body, or body stub: an object declaration, a type declaration (but not a task type declaration), an exception declaration, a renaming declaration, a number declaration, or a subtype declaration.

Implementation Guideline: Since these are all syntax errors, only one error should occur in each test. All combinations of declaration need not be checked.

Implementation Guideline: Treat body stubs for subprograms, packages, and tasks separately.

Check that a representation clause is not allowed after the declaration of a body (see IG 13.1/T6 and /T7).

- T5. Check that a subprogram declaration, single task declaration, task type declaration, use

clause, package declaration, generic declaration, and generic instantiation are allowed after the occurrence of a body.

Check that a representation clause can be given between two type declarations.

- T6. Check that PROGRAM_ERROR is raised if an attempt is made to call a subprogram whose body has not yet been elaborated. Check at least the following cases:

- a function is called in the initialization expression of a scalar variable or a record component, and the scalar or record variable's declaration is elaborated before the subprogram body is elaborated.

Implementation Guideline: Use a variable declared in a declarative part and in a package specification. Include a case of a generic package specification, and show that PROGRAM_ERROR is raised when the package is instantiated or the record type is used as an actual generic parameter to declare a variable.

- the subprogram (function or procedure) is called in a package body.

Implementation Guideline: Include premature use of a user-defined equality operator for a limited private type (except when "=" is declared by generic instantiation).

- the subprogram is an actual generic parameter called during elaboration of the generic instantiation.

- the subprogram is called during elaboration of an optional package body.

Implementation Guideline: Include a case where the body is separately compiled.

- a function is used in a default expression for a subprogram or a formal generic parameter; PROGRAM_ERROR is raised when an attempt is made to evaluate the default expression.

Implementation Guideline: In some tests, ensure that the elaboration check must be performed at run-time by constructing a case such that the same sequence of declarations and statements sometimes raises PROGRAM_ERROR when executed and sometimes does not.

For subprogram library units, check whether PROGRAM_ERROR is raised if the subprogram is called when another library unit is elaborated and pragma ELABORATE has not been used (see IG 10.5/T5).

Check that PROGRAM_ERROR is not raised if a subprogram's body (or body stub) has been elaborated before it is called. In particular, check that:

- a subprogram can appear in a non-elaborated declarative part or package specification before its body; no PROGRAM_ERROR is raised as long as the subprogram is called after its body has been elaborated.
- for a subprogram library unit used in another unit, no PROGRAM_ERROR is raised if pragma ELABORATE names the subprogram.

- T7. Check that PROGRAM_ERROR is raised if an attempt is made to instantiate a generic unit whose body has not yet been elaborated. Check at least the following cases:

Implementation Guideline: Use all forms of generic unit (package, procedure, and function). In the case of packages, use a package specification that requires a body unless the test objective specifies otherwise.

- a simple case where the generic unit body occurs later in the same declarative part.
- the generic unit is declared and instantiated in a package specification.
- a generic package has an optional body provided later in the same declarative part.
- a generic package has an optional body provided in a separately compiled unit.

Implementation Guideline: In some tests, ensure that the elaboration check must be performed at run-time by constructing a case so that the same sequence of declarations and statements sometimes raises PROGRAM_ERROR when executed and sometimes does not.

Check whether `PROGRAM_ERROR` is raised when a generic library unit instantiation is elaborated by another library unit and a pragma `ELABORATE` has not been used.

Implementation Guideline: Include a check using the predefined I/O library units.

Check that no `PROGRAM_ERROR` is raised by the elaboration of an instantiation if the body of the generic unit has been elaborated.

Implementation Guideline: Use cases similar to those for T6.

- T8. Check that `PROGRAM_ERROR` is raised when an attempt is made to activate a task before its body has been elaborated. In particular, check that `PROGRAM_ERROR` is raised:

- when a task "variable" is declared in a package specification and the package body occurs before the task body.
- when an allocator is evaluated before the task body has been elaborated.
- when several tasks are to be activated, and only some have unelaborated bodies.

Implementation Guideline: The task with an unelaborated body should be declared between tasks that will activate normally and tasks that raise an exception when their declarative parts are elaborated.

- when a generic unit is instantiated with a task that must be activated as part of the generic unit's elaboration.

Implementation Guideline: In some tests, ensure that the elaboration check must be performed at run-time by constructing a case so that the same sequence of declarations and statements sometimes raises `PROGRAM_ERROR` when executed and sometimes does not.

Check that a task with an unelaborated body can be called before it has been activated (see IG 9.5/T12).

Chapter 4

Names and Expressions

4.1 Names

Semantic Ramifications

S1. RM 3.1/1 lists the named entities defined by the language. Certain entities are declared *explicitly* by some specific construct; others are declared *implicitly* at a place (in the text) other than that where the corresponding identifier first appears (for example, a loop name is declared in the declarative part of the innermost enclosing block, subprogram, package, or task body (RM 5.1/3) instead of at the place where it appears).

named entity	declaration
number	number declaration
object	object declaration
discriminant	record type definition
record component	component declaration
loop parameter	iteration clause
exception	exception declaration
type	type declaration
subtype	subtype declaration
subprogram	subprogram declaration
package	package declaration
task unit	task declaration
generic unit	generic declaration
single entry	entry declaration
entry family	entry declaration
formal parameter	subprogram or entry declaration
generic formal param.	generic declaration
named block	implicit
named loop	implicit
labeled statement	implicit
enumeration literal	enumeration type definition
attribute	cannot be declared

Also, subprograms and enumeration literals can be declared implicitly by derivation (RM 3.4/11, /6).

S2. The following kinds of entities can be named by a simple name if and only if a suitable renaming declaration is provided:

- objects (including discriminants) that are components of other objects;
- entries of a family, or entries when named outside the body of the corresponding task; and
- functions denoted by attributes.

S3. Because a character literal is a name (RM 4.1/2) considered to denote a function (RM 3.5.1/3), it can be renamed in a renaming declaration as a function. Also, it can be used as the default name in a generic formal function declaration:


```
function TILDE return CHARACTER renames '~';
```

```
generic
```

```
with function TILDE return CHARACTER is '~';
```

S4. When the prefix of a selected component is the name of an enclosing function (whether or not the function has parameters), the prefix is considered a name and not a function call (RM 4.1.3/19). That is, evaluation of the prefix does not invoke the function, because the prefix is not considered a function call (see also IG 4.1.3/S for a fuller discussion):

```
type REC is
```

```
record
```

```
    X : INTEGER := 5;
```

```
end record;
```

```
function F (P : INTEGER := 4) return REC is
```

```
    X : INTEGER := 3;
```

```
    Y : INTEGER := F.X;      -- unambiguous; prefix used as name
```

```
    Z : INTEGER := F(2).X;   -- legal; function call is allowed
```

F(2).X is legal since F(2) is clearly a function call, not a name of the enclosing unit.

S5. ADDRESS, SIZE, POSITION, FIRST_BIT, LAST_BIT, and STORAGE_SIZE are representation attributes (RM 13.7.2). The prefix for any representation attribute except STORAGE_SIZE (RM 13.7.2/2, /4, /6, /7, and /13) can be an object that has an access type. If the value of such a prefix is null, CONSTRAINT_ERROR is not raised (RM 4.1/10).

Changes from July 1982

S6. If a prefix has an access type, the prefix must not denote an out parameter or a subcomponent of an out parameter.

Changes from July 1980

S7. The term "appropriate for a type" is introduced to clarify the rules regarding allowable prefixes in names.

S8. A character literal is considered a name.

Legality Rules

L1. If a prefix has an access type, it must not denote an out parameter or a subcomponent of an out parameter (RM 4.1/4).

Exception Conditions

E1. CONSTRAINT_ERROR is raised if a prefix has a null access value, except when the prefix is used for the attribute ADDRESS, SIZE, POSITION, FIRST_BIT, or LAST_BIT (RM 4.1/10).

Test Objectives and Design Guidelines

T1. Check that a prefix cannot denote an out parameter or a subcomponent of an out parameter if the prefix has an access type (see IG 6.2/T6).

T2. Check that CONSTRAINT_ERROR is raised if the prefix of a name has a null access value and the prefix is used in a name having the form of an indexed component, slice, selected component, or attribute (other than a representation attribute) (see IG 4.1.1/T5, IG 4.1.2/T5, IG 4.1.3/T4, and IG 4.1.4/T1).

- T3. Check that a character literal (or other enumeration literal) can be used as the default name in a formal generic function declaration (see IG 12.1.3/T1) and as the name in a renaming declaration for a function (see IG 8.5/T19).

4.1.1 Indexed Components

Semantic Ramifications

- S1. An array aggregate or an allocator is not syntactically a name and, therefore, may not be indexed. For example, the following are not allowed:

```
I := (3, 5, 7, 8, 16) (K);      -- illegal
I := new A' (3, 5, 7, 8, 15) (K); -- illegal
```

However, a function that returns an array or an access value can be an acceptable prefix, e.g., "&" ((3, 5, 7), (8, 15)) (K).

- S2. If the prefix of an indexed component is a call to an overloaded function, then overloading resolution is based on the actual parameter types, on the number of indexes given in the name, and on the types of the indexes (see IG 8.7.b/S). Consider the following example.

```
B : BOOLEAN := F(1, 2, 3);
```

The type of the result returned by F can be used in overloading resolution; an F that returns a three-dimensional boolean array (whose indexes have integer types) can be selected.

- S3. If the prefix of an indexed component is a variable or a constant, the name itself is a variable or a constant. (RM 3.2.1/2 says "a subcomponent of a constant is a constant." RM 3.2.1/3 says "an object that is not a constant is called a variable," and RM 3.2/7 says "an object is ... a component ... of another object," so a component of an array variable is a variable.) Similarly, a component of an array value is a value. (An array value is obtained by calling a function (RM 6/2). Although a function call returns a value, an indexed function call can be a variable (e.g., such an indexed component can be assigned to). This can happen when the call yields an access value that designates an array. Indexing the call means indexing the object designated by the access value. Such an object is a variable. Thus, indexing the result of a function call that designates an array yields a variable, not a value.)

- S4. The evaluation of an indexed component evaluates the index expressions and the prefix in an order not defined by the language. Since some order is chosen, this means that if any evaluable construct in a prefix is evaluated, then all such constructs must be evaluated before evaluating an index expression. For example:

```
... FUNC(F1, F2) .A(F3)
```

Assuming that F1, F2, and F3 are functions with side effects and F3 is an index expression, then the following evaluation orders are allowed or not, as noted:

```
F1, F2, F3      -- allowed
F2, F1, F3      -- allowed
F1, F3, F2      -- not allowed
F2, F3, F1      -- not allowed
F3, F1, F2      -- allowed
F3, F2, F1      -- allowed
```

Changes from July 1982

- S5. There are no changes.

Changes from July 1980

S6. The evaluation order of the index expressions and of the prefix is explicitly stated to be some order not defined by the language.

S7. If a prefix is a function call, **CONSTRAINT_ERROR** is raised if its value is null.

Legality Rules

- L1. The prefix of an indexed component must have either an array type or an access type whose designated type is an array type (RM 4.1.1/3 and RM 4.1/6-8), or the prefix must denote an entry family (RM 4.1.1/3).
- L2. For an indexed component whose prefix has an array type or an access type whose designated type is an array type, there must be one expression for each index position of the array type (RM 4.1.1/3).
- L3. For a prefix that denotes an entry family, there must be exactly one index expression (RM 4.1.1/3).
- L4. Each expression in an indexed component must have the base type of the corresponding index (RM 4.1.1/4).
- L5. If a prefix of an indexed component has an access type, the prefix must not denote an out parameter or a subcomponent of an out parameter (RM 4.1/4).

Exception Conditions

- E1. **CONSTRAINT_ERROR** is raised if the evaluation of an index expression gives an index value outside the range specified for the corresponding index of the array or entry family denoted (or designated) by the prefix (RM 4.1.1/4).
- E2. **CONSTRAINT_ERROR** is raised if the prefix of the indexed component has the value null (RM 4.1/10).

Test Objectives and Design Guidelines

- T1. Check that neither too few nor too many index values are accepted.

Implementation Guideline: Check for access types and array types. Entry families are checked in IG 9.5/T1.

Check that the base type of a subscript must be the same as the base type of the index.

Implementation Guideline: Check for access types and array types. Entry families are checked in IG 9.5/T1.

For indexed components whose prefix is a function call, check that the number of index values, the type of the index values, and the required type of the indexed component can be used to resolve an overloading of the prefix and of the index expressions (see IG 8.7.b/T23).

- T2. Check that the prefix cannot be:

- an aggregate;
- an allocator;
- an identifier denoting an access object whose value designates another access object whose value designates an array.

- T3. Check that the prefix may be:

- an identifier denoting an array object;
- an identifier denoting an access object whose value designates an array object;

- a function call delivering an array object;
Implementation Guideline: Include function calls using "&" and the logical operators.
- a function call delivering an access value that designates an array;
- a slice (check that both lower and upper bound components can be accessed);
- an indexed component denoting an array object (array of arrays);
- an identifier prefixed by the name of the innermost unit enclosing its declaration;
- a record component (of a record containing one or more arrays whose bounds depend on a discriminant).

Check that the appropriate component is accessed (see also T7).

- T4. Check that `CONSTRAINT_ERROR` is raised if an expression gives an index value outside the range specified for the index.

Implementation Guideline: Check for access types and array types. (For entry families, see IG 9.5/T8).

Implementation Guideline: Check for null arrays.

Implementation Guideline: Check for arrays whose bounds depend on the value of discriminants.

- T5. Check that `CONSTRAINT_ERROR` is raised if the prefix of an indexed component denotes an access object whose value is null, and also if the prefix is a function call delivering null.

- T6. Check that for suitably declared arrays, `SYSTEM.MIN_INT` and `SYSTEM.MAX_INT` can be used as subscripts.

Implementation Guideline: Declare small arrays that use these values as bounds.

- T7. Check that for an array having both positive and negative index values, the proper component is selected.

Check that for an array indexed with an enumeration type, appropriate components can be selected.

Check that subscript expressions can be of complexity greater than variable + - constant.

Check that multidimensional arrays are properly indexed.

Implementation Guideline: Use a slice of a `STRING`, among other cases, and use all parameter modes.

- T8. Check that expressions in the prefix are all evaluated either before or after the index expression.

4.1.2 Slices

Semantic Ramifications

S1. An array aggregate or an allocator creating an array object is not syntactically a name or a function call, and so cannot be used as the prefix in a slice.

S2. If the prefix of a slice is a call to an overloaded function, then overloading resolution uses: 1) the requirement that the prefix return a value having a one-dimensional array type or having an access type whose designated type is a one-dimensional array type; 2) the type of the discrete range; and 3) the required type of the slice itself (see IG 8.7.b/S).

S3. The evaluation of a slice evaluates the discrete range and the prefix in an order not defined by the language. Since some order is chosen, this means that the prefix must be completely evaluated before evaluating any expressions in the discrete range, or vice versa. For example:

```
... FUNC(F1, F2) .A(F3..F4)
```

Assuming that F1, F2, F3, and F4 are functions with side effects, then F1 and F2 must be evaluated before F3 and F4, or F3 and F4 must be evaluated first. It is not allowed to evaluate F1, for example, and then F3 or F4.

S4. In forming a null slice, the lower bound of the null discrete range must satisfy any constraints imposed by the *base type* of the index (RM 3.6.1/4 and RM 3.5/3). In particular, a null slice of a null array can be formed:

```
type A is array (INTEGER range <>) of BOOLEAN;
NULL_1 : A (5..4);
NULL_2 : constant A := NULL_1 (50..49);    -- no exception raised
```

S5. Since a slice has an array type, a slice can serve as the prefix for another slice, e.g.:

```
... B (1..19) (3..5) ...      -- equivalent to B (3..5)
... B (2..15) (4..10) (5..6) ... -- equivalent to B (5..6)
... B (1..10) (1..10) (1..10) ... -- equivalent to B (1..10)
... B (5..5) (5..5) (5) ...    -- equivalent to B (5)
```

If a null slice is given, it can be used as the prefix of another null slice.

Changes from July 1982

S6. The evaluation order of the prefix and the discrete range is explicitly stated to be some order not defined by the language.

Changes from July 1980

S7. There are no significant changes.

Legality Rules

- L1. The prefix of a slice must have a one-dimensional array type or an access type whose designated type is a one-dimensional array type (RM 4.1.2/3 and RM 4.1/6-8).
- L2. The discrete range of a slice must have the base type of the array index (RM 4.1.2/3).
- L3. If a prefix of a slice has an access type, the prefix must not denote an out parameter or a subcomponent of an out parameter (RM 4.1/4).

Exception Conditions

- E1. CONSTRAINT_ERROR is raised if the discrete range in a slice is not null and at least one bound of the discrete range is not in the range of index values for the array specified by the prefix (RM 4.1.2/4).
- E2. CONSTRAINT_ERROR is raised if the prefix has the value null (RM 4.1/10).

Test Objectives and Design Guidelines

- T1. Check that more than one discrete range is forbidden, even for multidimensional arrays.

Implementation Guideline: Try slicing on the first and last dimension; for example, when A is a two-dimensional array:

```
A (1..10)
A (1..10, 9)
A (5, 3..7)
A (1..10, 3..7)
```

Check that a single discrete range cannot be given when the prefix denotes a multidimensional array.

Implementation Guideline: These checks should be performed for access types and array types. Function calls delivering an array should also be used.

Check that entry families cannot be sliced.

Check that the base type of the discrete range must be the same as the base type of the index.

Implementation Guideline: Use one-dimensional arrays.

For slices whose prefix is a function call, check that the requirement for a one-dimensional array, the type of the discrete range, and the required type of the slice value can be used to resolve an overloading of the prefix or a bound of the discrete range (see IG 8.7.b/T24).

T2. Check that the prefix cannot be:

- an aggregate;
- an allocator;
- an identifier denoting an access object whose value designates another access object whose value designates an array.

T3. Check that the prefix may be

- an identifier denoting a one-dimensional array object;
- an identifier denoting an access object whose value designates a one-dimensional array object;
- a function call delivering a one-dimensional array object;
- a function call delivering an access value that designates a one-dimensional array;
- a slice;
- an indexed component denoting a one-dimensional array object (array of arrays);
- an identifier prefixed by the name of the innermost unit enclosing its declaration.
- a record component (of a record containing one or more arrays whose bounds depend on a discriminant).

Check that the prefix can have a limited (array) type.

- T4. Check that `CONSTRAINT_ERROR` is raised if the discrete range is not null and its upper or lower bound (check both cases separately) is not a possible index for the named array (see IG 3.6.1.a/T4).
- T5. Check that `CONSTRAINT_ERROR` is raised if the prefix of a slice denotes an access object whose value is null, and also if the prefix is a function call delivering null.
- T6. Check that a non-null range, L..R, can be used to form a null slice from an array A (see IG 3.6.1.a/T4).
- T7. Check that the discrete range in a slice can have the form A'RANGE (when A is either a constrained array type or an array object).

4.1.3 Selected Components

Semantic Ramifications

s1. The rules of the language guarantee that discriminants always have a value (see RM 3.7.2/8-12). Hence, a discriminant value can always be used to check the validity of references to components whose existence depends on the value of a discriminant.

s2. When the prefix of a selected component has an access type, it is not correct to think of the semantic effect as being equivalent to writing `.all` after the prefix. For example:

```

type REC is
  record
    X : INTEGER := 3;
  end record;
-- Implicit declarations include an operation for selection of
-- component X (the operation might be called ".X").
type ACC_REC is access REC;
-- Implicit declarations include an operation for selection of
-- component X (the operation might be called ".X").
-- Also, an operation to obtain an object designated by an
-- access value is declared here (call the operation ".all").
VA_REC : ACC_REC;
V_REC  : REC;
...
... VA_REC.X          -- Uses ".X" operation declared for ACC_REC.
... VA_REC.all.X      -- Uses ".all" operation declared for ACC_REC
--                      and ".X" operation declared for REC.
... V_REC.X           -- Uses ".X" operation declared for REC.

```

RM 3.7.4/1 specifies basic operations that are (according to RM 3.3.3/2) implicitly declared for a record type, and similarly, RM 3.8.2/1 specifies implicitly declared basic operations for access types. The wording in these sections specifies the operations by referring to the syntactic form used to invoke the operations, e.g., RM 3.8.2/1 says the operations "include the formation of a selected component with the reserved word `all`." RM 3.8.2/1 also says "if the designated type is a record type, [the operations] include the selection of corresponding components." The semantics of the component selection operations declared for an access type are specified in RM 4.1.3/11-12 (for `.all`) and RM 4.1.3/6-8 for selection of components through a selector. These paragraphs apply to the operations declared for access types because, in RM 4.1.3/11-12, the type of the prefix is required to be an access type, and in RM 4.1.3/6-8, the prefix is required to be appropriate for a record type (in particular, a record type having a component with the same identifier as that used in the selector of the selected-component name); "appropriate for a type" means the prefix can have an access type whose designated type is a record type with an appropriately named component (RM 4.1/8). The RM never names these operations, so it must use descriptive phrases that serve to link statements that the operations exist (in RM 3.7 and RM 3.8) with specifications of their semantics. In understanding the implications of the RM rules, it is helpful to give notations for the operations, as in the example above, where `".X"` is an overloaded notation denoting the component selection operations declared for both `REC` and `ACC_REC`. One might think of the name `VA_REC.X` as being equivalent to `".X"(VA_REC)`, and `VA_REC.all.X` as being equivalent to `".X"(".all"(VA_REC))`.

s3. The fact that `".X"` denotes different operations for an access type and for the designated type is important only when considering what operations are declared for private types in conjunction with RM 7.4.2/6-7 (see IG 7.4.2/S for a full discussion):

```

package P is
  type T is private;

  package Q is
    type ACC_T is access T; -- ".all" declared
  end;
private
  type T is new REC;      -- using the definition from above
    -- ".X" declared for T, but not for ACC_T (RM 7.4.2/8, /7)
  U : ACC_T := new T;
  V : INTEGER := U.all.X; -- legal; uses T's ".X"
  W : INTEGER := U.X;     -- illegal; no ".X" declared for ACC_T
end P;

```

According to RM 7.4.2/7-8, the ".X" operation for ACC_T is only declared within Q's body, and so is not visible within P's private part. Since the two notations are not both legal in the same contexts, U.X is not semantically equivalent to U.all.X.

S4. RM 4.1.3/17 says "The prefix must denote a construct that is either a program unit, a block statement, a loop statement, or an accept statement." This wording must be interpreted carefully for prefixes that denote bodies of task types. In particular, the prefix must not denote a task object. If an expanded name is used to refer to an entity declared in an enclosing task body, the prefix denotes the task body, not the task object:

```

task type T;
type T_INDEX is range 1..10;
A : array (T_INDEX) of T;
task body T is
  K : T_INDEX;
  X : INTEGER;
begin
  A(K).X := 5; -- illegal
  T.X := 5;    -- ok
end T;

```

The selected component A(K).X is illegal because A(K) denotes an object, not a unit. T.X is okay even though RM 9.1/4 forbids use of the simple name T as a type mark within the task body; T's use in T.X is not as a type mark but as the name of an enclosing unit.

S5. The selector of an expanded name must denote a declaration occurring "immediately within" the construct named by the prefix. Since blocks and loops need not have names, this means that entities declared immediately within a nameless block or loop cannot be named using an expanded name.

S6. Since labels are implicitly declared in the declarative part of the innermost enclosing block, subprogram body, task body, or package body (see RM 5.1/3), expanded name notation is available for labels. The prefix must be the name of the construct enclosing the implicit declaration, not the name of the construct enclosing the label's occurrence, e.g.:

```

procedure P is
begin
  BLK: begin
    LP: loop
      goto P.BLK.L; -- P.BLK.LP.L is illegal
    <<L>>...
  end loop LP;

```



```

    end BLK;
  end P;

```

Statement label L is implicitly declared within the innermost enclosing block or body, i.e., within block BLK. Since L is not declared within loop LP, the loop name cannot be used in an expanded name for L.

S7. A loop parameter is implicitly declared immediately within the loop (RM 5.5/6 and RM 8.1/6). Hence, if the loop is named, the loop name can be used as a prefix of the loop parameter identifier. Also, loop names are never declared within enclosing loops:

```

B1: begin
  L1: for I1 in 1..5 loop
    L2: for I2 in 1..5 loop
      ...
    end loop L2;
  end loop L1;
end B1;

```

Within L2, L2.I2 is a legal name for the loop parameter, as is B1.L2.I2. B1.L1.L2.I2 is illegal since L1 and L2 are both implicitly declared within block B1. B1.I2 is illegal, since I2 is implicitly declared within loop L2, not within block B1.

S8. Generic formal parameters are considered to be declared immediately within the generic unit, since a generic declaration forms a declarative region (RM 8.1/2) and the formal part of a generic declaration is contained within that region (RM 12.1/2). Consequently, if I is the name of a generic formal parameter for unit P, the expanded name P.I can be used within P's body even though the parameter textually precedes the occurrence of the generic unit's name (see AI-00412 and IG 12.1/S for further discussion):

```

generic
  I : INTEGER;
procedure P;

procedure P is
begin
  ... P.I ...    -- legal

```

S9. A name declared by a renaming declaration can be used as the selector in an expanded name, even though such a name does not declare an entity (it only declares another name for an entity; see RM 8.5/1) (see AI-00187):

```

package P is
  A : INTEGER;                -- declares an entity
end P;

with P;
package Q is
  B : INTEGER renames P.A;
end Q;

with Q;
package R is
  C : INTEGER renames Q.B;    -- okay
end R;

```

The declaration of R.C is legal even though B is not an entity declared within Q (B denotes P.A, an entity declared within P; R.C also denotes this entity).

S10. Consider the following use of a renamed package in an expanded name:

```
package P is
  X : INTEGER;
  package RENAMED_PACKAGE renames P;
  Y : INTEGER renames P.X;           -- legal
  Z : INTEGER renames RENAMED_PACKAGE.X -- legal (AI-00016)
end P;
```

RM 4.1.3/14-15 allows the expanded name given in Z's declaration since X is declared immediately within the visible part of a package. However, RM 4.1.3/18 says, "A name declared by a renaming declaration is not allowed as the prefix [of an expanded name used within the construct named by the prefix]." These rules conflict; AI-00016, however, says that if the selector of a expanded name is declared in the visible part of a package, the prefix can be a name declared by a renaming declaration.

S11. When using a name declared by a renaming declaration as the prefix of an expanded name, it is the context of the expanded name and the selector that determines whether the expanded name is legal, not the context where the prefix was declared. For example:

```
package LONG_P is
  X : INTEGER := 0;
end LONG_P;

package P renames LONG_P;

package body LONG_P is
  Y : INTEGER := 1;
  Z : INTEGER := P.Y;           -- illegal; Y not in visible part
  W : INTEGER := P.X;           -- legal; X in visible part
```

P.Y is illegal because P.Y is an expanded name, P is declared by a renaming declaration, and Y is not declared in LONG_P's visible part.

S12. An expanded name can be arbitrarily long if an enclosing package is renamed within its visible part:

```
package FRED is
  A : INTEGER;
  package JIM renames FRED;
private
  B : INTEGER;
end FRED;
```

Since A and JIM are declared in the visible part of package FRED and since a name declared by a renaming declaration can be used as the selector in an expanded name, it is possible to refer to FRED.A as FRED.JIM.A or FRED.JIM.- JIM.A, or in general, to use as many .JIM's as desired. However, inside package FRED, FRED.JIM.B is illegal, since a name declared by a renaming declaration is not allowed as a prefix in an expanded name except when the selector is declared in the visible part of a package (AI-00016 and RM 4.1.3/18). (Of course, whether inside or outside FRED, writing use FRED.JIM has the same effect as writing use FRED.)

S13. When the prefix of a selected component is a function name, the effect of evaluating the prefix depends on whether or not the name is an expanded name:

```

type REC is
  record
    X : INTEGER;
  end record;

function F return REC is
  X : INTEGER := 3;
  Z : INTEGER := F.X;      -- F is not called
begin ... end F;

A : INTEGER := F.X;        -- F is called

```

In the first use of F.X, F denotes an enclosing program unit, and so is considered an expanded name whose prefix has the syntactic form <name> instead of the form <function_call>. Hence, RM 4.1/9 applies, and the evaluation of the prefix just determines the construct enclosing the declaration of X. In the second usage, we assume there is no enclosing construct whose name is F, so F.X is not an expanded name. In this case, the only entity denoted by F is the function F, and since F.X is not an expanded name, the prefix must be considered to have the form <function call>. The evaluation of the prefix thus causes function F to be called (RM 4.1/10). In short, from the RM viewpoint, the effect of evaluating a prefix depends on how the prefix is parsed, and the parse depends on the context containing the name.

S14. Now consider a modification of the original example:

```

type REC2 is
  record
    Y : INTEGER;
  end record;

function F return REC is
  function F return REC2;
  X : INTEGER := 3;
  Z : INTEGER := F.X;      -- legal

```

This use of F.X is legal because F is the name of a unit enclosing the declaration of X, and although two Fs are visible, only one of the visible Fs is the name of an enclosing subprogram. Therefore, only the interpretation of F.X as an expanded name is considered (RM 4.1.3/19).

S15. If more than one visible enclosing subprogram (or accept statement) has the name of the prefix, then the name is illegal, independently of the selector (RM 4.1.3/18):

```

function F return INTEGER is
  X : INTEGER;

function F return FLOAT is
  Y : INTEGER := F.X;      -- illegal

```

F.X is illegal because both enclosing subprograms are visible. It does not matter that X is only declared within one of the enclosing units.

S16. If two subprograms have identical designators as well as the same parameter and result type profile, then the outer subprogram is not visible within the inner subprogram (RM 8.3/15):

```

function F return REC is
  X : INTEGER := 3;

```

```
function F return REC is
```

```
  X : INTEGER := 3;
```

```
  Y : INTEGER := F.X;      -- legal expanded name
```

F.X is legal because the only visible F is the innermost F.

S17. Now let's combine some of these cases:

```
function F return T1 is
```

```
-- F1
```

```
  X : INTEGER;
```

```
  function G return T2 is
```

```
    X : INTEGER;
```

```
    function F return T2 renames G;      -- F2
```

```
    -- note F2 renames an enclosing unit
```

```
    Y : INTEGER := F.X;                  -- legal? which F?
```

If T1 and T2 denote different types, then two Fs are visible. RM 4.1.3/18 says "if the prefix is the name of a subprogram or accept statement and if there is more than one visible enclosing subprogram or accept statement of *this name* [emphasis supplied], the expanded name is ambiguous, independently of the selector." Although two Fs are visible and each F *denotes* an enclosing unit, only one enclosing subprogram is "of the name" given as the prefix, so the prefix is legal and denotes F1. (If the RM had mentioned enclosing units "denoted by the name given as the prefix," F.X would have been illegal.)

If T1 and T2 denote the same type, the declaration of F2 hides the declaration of F1, so only one F is visible at F.X. However, since the only visible F is declared by a renaming declaration and since F denotes an enclosing unit, F.X is an expanded name; therefore, RM 4.1.3/18 applies, and F.X is illegal.

S18. In short, when a subprogram designator is used as a prefix, an implementation must determine whether more than one visible enclosing subprogram has the same designator. If so, the name is illegal. If there is only one visible enclosing subprogram, then the prefix is considered to denote the enclosing unit and is not called. If there is no such enclosing unit, then the prefix must be interpretable as a function call. Of course, if the prefix can only be parsed as a function call, then that is the interpretation used:

```
function F (X : INTEGER) return REC is
```

```
  X : INTEGER := 3;
```

```
function F return REC is
```

```
  W : INTEGER := F(3).X;      -- legal
```

```
  Z : INTEGER := F.W;         -- illegal
```

F(3) is unambiguously a function call (since the parameterless F does not return an array type and so cannot be indexed). Since F(3) is a function call, F(3) cannot be a prefix of an expanded name. That there are two visible enclosing units called F is irrelevant. This fact is, however, relevant to F.W, since in this case, F can be interpreted as the name of an enclosing unit. RM 4.1.3/19 says this interpretation is preferred, and then RM 4.1.3/18 says the prefix is illegal because there are two visible enclosing Fs. Similarly, if the inner F returned an access value, then F.all would be illegal within the inner function.

Approved Interpretations

S19. An expanded name denoting a generic formal parameter is allowed within a generic unit (AI-00412).

A simple name declared immediately within the visible part of a generic package specification

can be the selector of an expanded name occurring within the generic unit if the prefix of the expanded name is declared by a renaming declaration and denotes the generic package (AI-00412).

S20. A simple name declared in the visible part of a package specification can be the selector of an expanded name whose prefix denotes the package and is a name declared by a renaming declaration (AI-00016).

S21. A name declared by a renaming declaration can be used as the selector in an expanded name (AI-00187).

Changes from July 1982

S22. A name declared by a renaming declaration is not allowed as the prefix of an expanded name.

S23. If according to the visibility rules, the prefix of a selected component can be interpreted as the name of an enclosing subprogram or an accept statement, no interpretations as a function call are considered.

Changes from July 1980

S24. The selector in an expanded name can be a character literal.

S25. Expanded names can be used within the body of an accept statement if the prefix denotes the enclosing single entry or entry family.

S26. A prefix can be an access value designating a task object when the selector is an entry.

Legality Rules

- L1. For a name of the form L.R, if L can only be parsed as a function call or if there is no enclosing unit denoted by L, then
 - a. if L has a record type or an access type whose designated type is a record type, R must be the identifier of one of the record's components (RM 4.1.3/4-7);
 - b. if L has a task type or an access type whose designated type is a task type, R must be an entry, or an entry family, declared in the corresponding task or task type declaration (RM 4.1.3/9-10);
 - c. if R is the reserved identifier all, L must have an access type (RM 4.1.3/11-12); and
 - d. if L denotes a package, R must be declared immediately within the visible part of the package (RM 4.1.3/14-15 and AI-00187).
- L2. For a name of the form L.R occurring inside a unit denoted by L (where the unit may be a subprogram body, task body, accept statement (for single entry L or entry family L), package specification or body, loop, or block (RM 4.1.3/17)):
 - L must not be declared by a renaming declaration (RM 4.1.3/18);
 - L must be the only such visible enclosing unit (RM 4.1.3/18);
 - R must be an identifier, a character literal, or an operator symbol declared immediately within L (RM 4.1.3/17).
- L3. If a prefix of a selected component has an access type, the prefix must not denote an out parameter or a subcomponent of an out parameter (RM 4.1/4).

Exception Conditions

- E1. **CONSTRAINT_ERROR** is raised by the evaluation of a selected component of the form **L.R**, where **R** is an identifier, if **L** has
- a record type with variants, or
 - an access type whose designated type is a record type with variants,
- and **R** is the identifier of a component that does not exist for the current discriminant values of the record object (RM 4.1.3/8).
- E2. **CONSTRAINT_ERROR** is raised by the evaluation of a selected component of the form **L.R**, if **L** has the value null (RM 4.1/10).

Test Objectives and Design Guidelines

- T1. Check that the notation **L.R** may be used to denote a record component (including a discriminant), where **R** is the identifier of such a component, and **L** may be any of the following:

- an identifier denoting a record object;
- an identifier denoting an access object whose value designates a record object;
- a function call delivering a record value;
- a function call delivering an access value designating a record;

Implementation Guideline: In the above four cases, reference components of variant records, and use components selected on both sides of an assignment statement and as a parameter of various modes.

- an indexed component;
- an identifier prefixed by the name of the innermost unit enclosing the identifier's declaration.
- a selected component denoting a record (which is a component of another record).

Implementation Guideline: In some cases, **L** should have a limited type and a generic formal or private type with discriminants.

- T2. Check that **L.R** is illegal if **L** is an aggregate or an allocator for a record type containing component **R**.
- T3. Check that the notation **L.all** is allowed if **L** is the name of an access object designating:
- a record object
 - an array object
 - a scalar object
 - another access object

Check that if **A** is an identifier denoting an access object that in turn designates an access object, the form **A.all.all** is accepted.

Implementation Guideline: Use both sides of the assignment statements and parameters of each mode.

- T4. Check that **L.R** raises **CONSTRAINT_ERROR** if:

- L is an access object or a function call having the value null;
- L is a record object or a function call returning a record value such that, for the existing discriminant values, the component denoted by R does not exist.
- L is an access object (or a function call returning an access value), and the object designated by the access value is such that component R does not exist for the object's current discriminant values.

T5. Check that if L is an operator symbol, L.R must occur inside a function declaring L.

T6. Check that if T has a task type or an access type designating a task type with entry E or entry family E, T.E is allowed as an entry call.

Implementation Guideline: See IG 9.5/T for cases where T has a task type. It is only necessary to test here cases where T has an access type that designates a task and cases where T is a function call.

T7. Check that if L is a package declared by a renaming declaration, D is the package denoted by L, and R is a name declared immediately within the visible part of D:

- L.R is legal whether L.R occurs outside D, inside D's visible part, inside D's private part, or inside D's body;
- L.R is illegal if R is declared immediately within the private part or body of D.

Implementation Guideline: Include (in a separate test) a case where R is declared immediately within D as a renaming of L and the full name is L.R.R.X.

Check that if R is declared immediately within a package, a generic package, a subprogram, a generic subprogram, a task, a block, a loop, or an accept statement named L, L.R is allowed inside the unit (L is not declared by a renaming declaration).

Implementation Guideline: R can be a loop parameter, a label, a block name, a loop name, or a generic formal parameter.

Implementation Guideline: For the accept statement, include both a single entry and an entry family case.

Implementation Guideline: Include cases where R is declared by a renaming declaration.

Implementation Guideline: For the subprogram, generic subprogram, and accept statement cases, there should be no enclosing visible subprogram or accept statement that is also named L.

T8. Check the following cases where F is the name of a function returning a record with component X and X is also declared within F:

- F.X occurs within F (the function should not be called).
- F.X occurs within F, F is declared within an enclosing subprogram or single entry accept statement that is also named F, and both Fs are visible (F.X is illegal).
- F.X occurs within F, F is declared within an enclosing function that is also named F, and only the innermost F is visible (the function should not be called).
- F.X occurs within F, F is declared within an enclosing subprogram or accept statement that is also named F, and both Fs are visible (F.X is illegal).
- F.X occurs within F and F is declared within an enclosing package, task, or accept statement for an entry family that is also named F (the function should not be called).
- the prefix has a parameter list so it can only be parsed as a function call; F(...).X occurs within F; F is declared within an enclosing subprogram or single entry accept statement that is also named F, and both Fs are visible (F is invoked).

- F.X occurs within F and another subprogram named F is declared immediately within F:
 - the inner declaration of F is not a renaming declaration, and
 - the inner F does not hide the outer F (F is not invoked).
 - the inner F hides the outer F (the inner F is invoked).
 - the inner declaration of F is a renaming declaration, and
 - the inner F does not hide the outer F (F is not invoked).
 - the inner F hides the outer F (F.X is illegal).

Check that F.all is illegal within a parameterless function F that returns an access value if the inner F is enclosed by another visible subprogram named F.

Implementation Guideline: Repeat the above checks when F is the name of a generic function.

- T9. Check that an expanded name is allowed even if a use clause has made it unnecessary to write an expanded name.
- T20. For an enumeration type declared in the visible part of a package P (including when P is a generic instance), check that the following implicitly declared entities can be selected from outside the package using an expanded name whose prefix is P: the enumeration literals (including character literals), the relational operators.
- Implementation Guideline:* Include a check for an overloaded enumeration literal.
- T21. For a derived boolean type declared in the visible part of a package P (including when P is a generic instance), check that the following implicitly declared entities can be selected from outside the package using an expanded name whose prefix is P: TRUE, FALSE, the relational operators, and the logical operators: "and", "or", "xor", and "not".
- T22. For an integer type declared in the visible part of a package P (including when P is a generic instance), check that the following implicitly declared entities can be selected from outside the package using an expanded name whose prefix is P: relational operators and arithmetic operators (unary and binary + and -, *, /, mod, rem, **, and abs).
- T23. For floating point type declared in the visible part of a package P (including when P is a generic instance), check that the following implicitly declared entities can be selected from outside the package using an expanded name whose prefix is P: relational operators and arithmetic operators (unary and binary + and -, *, /, **, and abs).
- T24. For fixed point type declared in the visible part of a package P (including when P is a generic instance), the following implicitly declared entities can be selected from outside the package using an expanded name whose prefix is P: the relational operators, * with one INTEGER operand, / with the second operand of type INTEGER, unary and binary + and -, and abs.

Check that the operations for multiplying or dividing two fixed point values are not implicitly declared in the unit, i.e., check that these operators cannot be named by selection using a prefix that denotes the unit (see RM 4.5.5/9). Also, check that there is no / operator whose first operand has type INTEGER and whose second is a fixed point type.

- T25. For an array type declared in the visible part of a package P (including when P is a generic instance), check that the following implicitly declared entities can be selected from outside the package using an expanded name whose prefix is P if:
- the component type is not limited: equality and inequality.

- the array has one dimension and the component type is:
 - non-limited: catenation, equality, inequality.
 - discrete: relational operators.
 - boolean: "not" and the logical operators.

Check that the above operations are not provided when the component type is not the required type.

- T26. For an access type declared in the visible part of a package P (including when P is a generic instance), check that the following implicitly declared entities can be selected from outside the package using an expanded name whose prefix is P: equality and inequality.
- T27. For a private type declared in the visible part of a package P (including when P is a generic instance), check that the following implicitly declared entities can be selected from outside the package using an expanded name whose prefix is P: equality and inequality if the type is not limited; no such operators if the type is limited.
- T28. For a derived type declared in the visible part of a package P (including when P is a generic instance), check that the following implicitly declared entities can be selected from outside the package using an expanded name whose prefix is P: derived subprograms.

4.1.4 Attributes

Semantic Ramifications

S1. This section discusses general properties of attributes. Specific properties of each attribute are discussed in the sections corresponding to the RM sections where each attribute is defined.

S2. Syntactically, an attribute such as A'LAST(3) can be parsed either as a function_call whose name is A'LAST and whose actual parameter is 3, or as an attribute whose prefix is A and whose attribute_designator is LAST(3). This syntactic ambiguity has no practical consequences. Attributes are shown in the RM with an optional single parameter for expository purposes, since some attributes are defined to require a single parameter (having a *universal_integer* type, e.g., 'RANGE). The attributes PRED, SUCC, POS, VAL, IMAGE, and VALUE are considered functions; CHARACTER'SUCC('A') is a function call and is allowed even though the argument is not a static *universal_integer* expression.

S3. The attribute designator is not considered when resolving a prefix. Consider the following example:

```
type A1 is array (1..10, BOOLEAN) of ... ;
type A2 is array (1..10) of ... ;
function F (X : INTEGER) return A1;
function F (X : INTEGER) return A2;
...
if F(3)'LAST(2)           -- illegal
```

The fact that the prefix of 'LAST(2) must denote a two-dimensional array (or an access type designating a two-dimensional array) or the fact that the name F(3)'LAST(2) must have the type BOOLEAN cannot be used in deciding which F is denoted in the prefix.

S4. The "meaning" of the prefix of an attribute is determined by the visibility rules (RM 8.3/1). For example:

```

type ACC is access INTEGER;
function F return ACC is
  X : SYSTEM.ADDRESS := F' ADDRESS;    -- legal?

```

The meaning of the prefix of 'ADDRESS is to be determined independently of the fact that it is the prefix of an attribute (RM 4.1.4/3), i.e., the declaration denoted by F is decided independently of the attribute designator. Clearly in this case, there is only one visible F, so F's use as a prefix is unambiguous. Now, however, we have to decide whether the prefix is to be parsed as a function call or simply as the name of a unit. In making this decision (which, technically speaking, does not affect the "meaning" of the prefix since it does not affect which declaration F denotes), we are not forbidden from looking at the attribute designator. In this case, RM 13.7.2/6 explicitly states that "if the prefix is the name of a function, the attribute is understood to be an attribute of the function (not of the result of calling the function)." This means that F is parsed as a name, not as a function call, and so evaluation of the prefix does not entail invoking the function.

S5. The nature of the "meaning" of a prefix is further illustrated by this example (suggested by R. S. Kotler):

```

type ACC_STRING is access STRING;
function F (X : INTEGER) return ACC_STRING;    -- F#1
function F return ACC_STRING;                  -- F#2
...
... F (3) ' ADDRESS                            -- F#2 (3) ' ADDRESS

```

The term "meaning" is defined in RM 8.3/1:

The meaning of the occurrence of an identifier at a given place in the text is defined by the visibility rules and also, in the case of overloaded declarations, by the overloading rules.

(The prefix F(3) does not have a meaning in the sense of RM 8.3/1 since it is not an identifier, but we can understand RM 4.1.4/3's use of the phrase "meaning of the prefix" as entailing "the meaning of the identifiers in the prefix".)

S6. RM 8.7/2 continues:

For overloaded entities, overload resolution determines the actual meaning that an occurrence of an identifier has, whenever the visibility rules have determined that more than one meaning is acceptable at the place of this occurrence. ...

S7. When more than one declaration of an identifier is visible, overloading resolution is required. RM 8.7/7 says:

When considering possible interpretations of a complete context, the only rules considered are the syntax rules, the scope and visibility rules, and the rules of the form described below [in RM 8.7/8-15].

In particular, note that "the syntax rules" must be used in deciding what the interpretation (i.e., meaning) of an identifier is.

S8. Now let's consider the effect of these rules on some examples that are simpler than the one presented originally. First, consider:

```

type ACC_STRING is access STRING;
procedure G (X : INTEGER) is ... end;
function G (X : INTEGER) return ACC_STRING is ... end;

... G (3) ' LAST ...      -- legal
... G (3) ' SIZE ...      -- legal

```

First, it is clear that the visibility rules alone give two "meanings" for G, so overloading resolution must be attempted. Overloading resolution takes the syntax rules into account. In this case, the syntax rules require that G(3) be parsed as a prefix, and a prefix must be either a name or a function call. Since the syntax rules forbid an interpretation of G(3) as a procedure call, G is uniquely determined to denote function G. Note that in making this determination, we have not used any information about the attribute designator nor any information stemming from the use of G(3) as the prefix of an attribute; we have only used the fact that G(3) must be a prefix.

S9. In short, overloading resolution of the prefix of an attribute can take into account the fact that only a name or a function call is allowed, but it cannot take into account the nature of the attribute designator. If the identifiers in the prefix are unambiguous given the fact that they are being used in a prefix, the "meaning" of the identifiers has been determined. At this point, the attribute designator can be used to decide whether the prefix is to be parsed as a function call or not. (Since the meaning of identifiers in the prefix has been determined, the choice of a parse cannot further affect the "meaning" of the prefix.)

S10. For example, consider:

```
function H return ACC_STRING is ... end;

... H'LAST ...           -- legal
... H'ADDRESS ...        -- legal
```

Here H unambiguously denotes a specific function, so the "meaning" of H is unambiguous, but consideration of the prefix alone is insufficient for deciding whether H should be parsed as a name or as a function call. However, once the meaning has been uniquely determined, RM 4.1.4/3 does not forbid using the attribute designator to decide how the prefix should be parsed. For H'LAST, RM A/21 requires that the prefix be appropriate for an array type, i.e., the prefix must have either an array type or an access type whose designated type is an array type. This means the prefix cannot be the name of a function, but it can be a function call, so H'LAST is unambiguous (H is called). Similarly, for H'ADDRESS, RM 13.7.2/6 requires that:

if the prefix is the name of a function, the attribute is understood to be an attribute of the function (not of the result of calling the function).

Since the prefix in this case cannot be a function call, there is no ambiguity in deciding how to parse the prefix (H is not called), and so H'ADDRESS is legal.

S11. This example shows how the attribute designator can be taken into account when deciding how the prefix is to be parsed even though the designator cannot be used to help resolve the "meaning" of names occurring in the prefix.

S12. Now let's use this approach in considering the legality of F(3)'ADDRESS in the original example. F in the prefix has two meanings according to the visibility rules (i.e., F can denote either F#1 or F#2). If we try to resolve the meanings by taking into account the fact that F(3) is a prefix, we find that F(3) can be parsed either as a function call or as a name, depending on which declaration F is considered to denote. This means F(3) is ambiguous even after taking into account the fact that it is a prefix. Since the prefix is ambiguous, F(3)'ADDRESS is ambiguous; (RM 4.1.4/3 forbids using the attribute designator to help resolve the "meaning" of identifiers in the prefix). The fact that F(3)'ADDRESS is only legal if F is considered to denote F#2 is irrelevant because RM 4.1.4/3 does not allow us to use this information.

S13. The rule in RM 4.1.4/3 is not equivalent to saying that the prefix of an attribute is a "complete context." If the prefix were considered a complete context, we could use the attribute designator to help determine how the prefix is to be parsed. Since rules affecting the parse of a prefix are considered syntax rules, we would be allowed to use such rules to resolve the meaning of F(3) in the original example, i.e., F(3)'ADDRESS would be unambiguous.

S14. If the prefix of an attribute is allowed to be a function call and the prefix is overloaded, the prefix is ambiguous, regardless of the attribute designator. For example, if the prefix of 'CALLABLE is a function returning a task type and a function returning an array type, the prefix is ambiguous.

S15. Only the following attributes are allowed to have function calls as prefixes: 'CALLABLE, 'FIRST, 'FIRST(N), 'LAST, 'LAST(N), 'LENGTH, 'LENGTH(N), 'RANGE, 'RANGE(N), 'TERMINATED. In particular, although the prefixes of 'ADDRESS, 'SIZE, 'CONSTRAINED, and 'STORAGE_SIZE can be objects, they cannot be function calls.

S16. If the prefix of an attribute contains a function call, the function must be evaluated, e.g., consider:

A (F (X)) ' SIZE

where A is an array and F is a function. Even though the size of an array component may be the same for each component, the prefix (i.e., F(X)) must be evaluated (RM 4.1/10).

S17. Since T'IMAGE produces a value of type STRING, which has an array type, this attribute can be used as the prefix of an array attribute, e.g., T'IMAGE(N)'LAST is legal.

Changes from July 1982

S18. For purposes of overloading resolution, the meaning of a prefix must be determined independently of the attribute designator and independently of the fact that it is the prefix of an attribute.

Changes from July 1980

S19. An attribute can no longer be an exception.

S20. An attribute can now be a range.

S21. Implementation-defined attributes cannot have the same identifier as that of a predefined attribute.

Legality Rules

- L1. The prefix of 'SIZE and 'ADDRESS must not be a function call (RM 13.7.2/6).
- L2. The prefix of 'STORAGE_SIZE must not be a function call (RM 13.7.2/13).
- L3. The prefix of 'CONSTRAINED must not be a function call (RM A/6-7).
- L4. If a prefix of an attribute has an access type, the prefix must not denote an out parameter or a subcomponent of an out parameter (RM 4.1/4).

Exception Conditions

- E1. CONSTRAINT_ERROR is raised if the prefix has the value null and the attribute designator is not SIZE, ADDRESS, POSITION, FIRST_BIT, or LAST_BIT (RM 4.1/10).

Test Objectives and Design Guidelines

Each attribute is checked individually in the section where it is defined.

Overloading resolution for prefixes of attributes is checked in IG 8.7.b/T26.

- T1. Check that CONSTRAINT_ERROR is raised if the prefix of the following attributes has the value null:

- 'CALLABLE, 'TERMINATED: the designated type must be a task type.

- 'FIRST, 'FIRST(N), 'LAST, 'LAST(N), 'LENGTH, 'LENGTH(N), 'RANGE, 'RANGE(N): the designated type must be an array type.

Implementation Guideline: The prefix should be either an object or a function call.

- T2. Check that CONSTRAINT_ERROR is not raised if the prefix of 'ADDRESS, 'SIZE, 'FIRST_BIT, 'LAST_BIT, and 'POSITION has the value null.

Implementation Guideline: For 'ADDRESS and 'SIZE, it will suffice for the prefix to be an access object with any designated type.

Implementation Guideline: For 'FIRST_BIT, 'LAST_BIT, and 'POSITION, the prefix must be a record component.

- T3. Check that the prefix of 'SIZE and 'ADDRESS is evaluated even if the value does not seem to be needed.
- T4. Check that the IMAGE attribute can be a prefix of an array attribute, e.g., T'IMAGE(N)'FIRST.

Check that a qualified expression is not allowed as a prefix, e.g., STRING'(N)'LAST and similar forms are illegal.

4.2 Literals

Semantic Ramifications

S1. Real literals are often implicitly converted to some real type (RM 4.6/15). Such conversions must be exact if the real literal is a model number (RM 4.5.7/6 and IG 4.5.7/S). Note that even if 'DIGITS for a floating point type is N, the exact decimal representation of 'LARGE will require more than N digits, and this decimal value must be converted exactly to 'LARGE. For example, if N = 6, 'LARGE is 19_342_803_890_462_029_940_523_008.

S2. The lower bound of a string literal is determined by the context in which the literal appears. There are two kinds of context. In one, the context determines an unconstrained array subtype, and the lower bound is determined by 'FIRST of the index subtype:

```
subtype I5_10 is INTEGER range 5..10;
type S1 is array (I5_10 range <>) of CHARACTER;
CS1 : constant S1 := "A";    -- S1'FIRST = 5
procedure P (S : S1);
... P("A"); ...              -- within P, S'FIRST = 5
```

If 'FIRST of the index subtype is also 'FIRST of the index base type, then CONSTRAINT_ERROR will be raised for all null string literals of the type when they occur in an unconstrained context:

```
type ENUM is (A, B);
type S2 is array (ENUM range <>) of CHARACTER;
CS2 : constant S2 := "";      -- CONSTRAINT_ERROR
CS3 : S2 (B..A) := "";        -- no exception
CS4 : S2 (B..A) := "" & "";    -- CONSTRAINT_ERROR
... "" = S2'("A") ...        -- CONSTRAINT_ERROR
```

The string literal in the initialization of CS2 is given the lower bound A; CONSTRAINT_ERROR is raised when computing the upper bound, since A has no predecessor. No exception is raised by the initial value of CS3 since the lower bound is determined by the index constraint, and the lower bound, B, has a predecessor. CONSTRAINT_ERROR is raised by the initialization expression for CS4 since the null string literals occur in a unconstrained context -- the operands

of "&" are associated with an unconstrained formal parameter, and so, the upper bound of "" is undefined. Similarly, CONSTRAINT_ERROR is raised for the null string literal operand of the equality operator.

S3. In the following contexts, the lower bound of a string literal is determined by the index constraint associated with a constrained array type. (The source of the lower bound information is given in parentheses.)

- the expression of an assignment statement (the lower bound of the target variable);
- the expression giving the initial value of a variable or record component (the lower bound of the variable or component);
- the expression giving the initial value of a constant object when the subtype indication specifies a constrained array type (the lower bound of the subtype indication);
- the expression giving the default value of a subprogram, entry, or generic formal in parameter or the return value of a function when the parameter's or function's type mark denotes a constrained array subtype (the lower bound of the type mark);
- the expression in an aggregate specifying the value of an array or record component (the lower bound of the corresponding array or record component; note that an array or record component is necessarily constrained);
- the innermost subaggregate of a multidimensional array aggregate (see RM 4.3.2/2) when the multidimensional aggregate appears in one of the above contexts (the lower bound of the last dimension of the applicable constrained array subtype); see IG 4.3.2/S for further discussion.

S4. In the following contexts, the lower bound of a string literal is given by 'FIRST of the index subtype when the literal appears as:

- the expression giving the initial value of a constant object whose subtype indication denotes an unconstrained array type;
- an actual in parameter in a subprogram or entry call, when the formal parameter is unconstrained (note that this includes use as an operand of the predefined relational and catenation operators);
- the actual parameter in a generic instantiation when the formal parameter is unconstrained;
- the return expression in a function whose return type is unconstrained;
- the operand in a qualified expression, when the type mark denotes an unconstrained array type;
- the expression in a membership test;
- an expression enclosed in parentheses;
- the innermost subaggregate of a multidimensional array aggregate appearing in one of the above contexts.

Note that CONSTRAINT_ERROR will be raised whenever a null string literal appears in one of the above contexts and the lower bound of the index subtype is also the first value of the index base type, since in this case, the lower bound has no predecessor. CONSTRAINT_ERROR will

be raised for the same reason when a null string literal is used as a subaggregate of a multidimensional aggregate appearing in a context imposing an index constraint:

```
type A2 is array (INTEGER range <>,
                  INTEGER range <>) of CHARACTER;
X2 : A2 (3..2, INTEGER'FIRST..INTEGER'LAST) := (3..2 => "");
```

The applicable lower bound is given by the second discrete range, INTEGER'FIRST..INTEGER'LAST, and since INTEGER'FIRST has no predecessor, CONSTRAINT_ERROR is raised. Note that no exception is raised by the following declaration:

```
X3 : A2 (3..2, 1..10) := (3..2 => "");
```

The lower bound of "" is 1, which has a predecessor. The bounds 1..0 are then discarded by the implicit subtype conversion performed for the assignment.

s5. A string literal can have different lower bounds in a record aggregate even though it occurs only once. For example:

```
type R is
  record
    C1 : STRING (1..2);
    C2 : STRING (4..5);
  end record;
V : R := (C1 | C2 => "ab");
```

The aggregate is evaluated as though it had been written (AI-00244):

```
(C1 => "ab", C2 => "ab")
```

Hence, the literal for C1 has lower bound 1 and the literal for C2 has lower bound 4.

s6. The fact that null has an access type can be used in resolving its type:

```
type AI is access INTEGER;
type AF is access FLOAT;
procedure P (X : AI);
procedure P (X : INTEGER);
...
P(null);      -- unambiguous
```

The call is unambiguous because there is only one visible P that takes a parameter of an access type. Once P is resolved, the type of null is determined to be AI.

s7. The following examples show how overloading resolution can use the fact that a string literal is a one-dimensional array of character type:

```
type A2 is array (1..10, 1..5) of CHARACTER;
procedure P1 (X : A2);
procedure P1 (X : STRING);
... P1 ("AB");      -- unambiguous because of dimensionality
```

The fact that "AB" is a one-dimensional array can be used to resolve the call. Note that P (('A', 'B')) would be ambiguous since dimensionality information cannot be used to resolve the type of an aggregate (RM 4.3/7).

s8. Similarly, one can use the fact that a string literal has an array type.

```

type R1 is record null; end record;
procedure P2 (X : R1);
procedure P2 (X : STRING);

... P2 ("AB");           -- unambiguous because "AB" is an array type
... P2 (('A', 'B'));     -- ambiguous

```

The second call is ambiguous because there are two composite types in scope, so ('A', 'B') can be either one of these types, and P2 accepts parameters of either type.

S9. Finally, the class of the component type can be considered:

```

type AC is array (1..10) of INTEGER;
procedure P3 (X : AC);
procedure P3 (X : STRING);
... P3 ("AB");           -- unambiguous because of component type
... P3 (('A', 'B'));     -- ambiguous

```

S10. A one-dimensional array of a character type can be declared such that there are no usable string literals for the type except the null string literal:

```

type NO_CHAR is new CHARACTER range ASCII.NUL .. ASCII.BEL;
-- no graphic characters
type NO_LIT is array (POSITIVE range <>) of NO_CHAR;
X : NO_LIT (1..0) := "";
Y : NO_LIT (1..3) := "ABC"; -- CONSTRAINT_ERROR
procedure P4 (X : STRING);
procedure P4 (X : NO_LIT);
...
P4 ("DEF"); -- ambiguous

```

"ABC" raises CONSTRAINT_ERROR because it is equivalent to ('A', 'B', 'C'), and none of these character literals belong to the NO_LIT component subtype. Note that since NO_CHAR is derived from CHARACTER, NO_CHAR is a character type (RM 3.4/4), and all the character literals belong to NO_CHAR's base type. Hence, NO_LIT has the basic operation for forming string literals, and "ABC" is legal as a string literal of type NO_LIT.

S11. Character literals can fail to be visible when a character type is declared in a package:

```

package P is
  type C is new CHARACTER;
  type AC is array (1..2) of C;
end P;

X : P.AC := "AB"; -- illegal

```

Unless use P is written before the declaration of X, C's character literals are not visible, so "AB" is illegal.

S12. In addition, character literals can fail to exist for a type:

```

type CH is ('A', 'B');
type ACH is array (1..2) of CH;
X : ACH := "BC"; -- illegal

```

The literal 'C' is not visible for type CH since it is not a value of the type.

Changes from July 1982

S13. The upper bound of a null string literal is defined; `CONSTRAINT_ERROR` is raised when the bound does not exist.

S14. The character literals used in a string literal must be visible at the point where the string literal is used.

Changes from July 1980

S15. Rules for determining the bounds of a string literal are given.

S16. Rules are given for determining the type of a string literal and a null literal.

Legality Rules

L1. The type of the literal `null` must be determinable solely from the context in which the literal appears, but using the fact that the literal is a value of some access type (RM 4.2/4).

L2. The type of a string literal must be determinable solely from the context in which the literal appears, but using the fact that the literal is a value of a one-dimensional array type whose component type is a character type (RM 4.2/4).

L3. The character literals corresponding to the graphic characters contained within a string literal must be visible at the place where the string literal appears (RM 4.2/5).

Exception Conditions

E1. `CONSTRAINT_ERROR` is raised for a null string literal if the lower bound is 'FIRST of the index {base type (RM 4.2/3).

E2. `CONSTRAINT_ERROR` is raised for a string literal if any character in the literal does not belong to the component subtype (RM 4.2/3 and RM 4.3.2/11).

Test Objectives and Design Guidelines

T1. Check that real literals are converted with the required accuracy to a floating or fixed point value (see IG 2.4.1/T2).

T2. Check that the type of the literal `null` is determined solely from the context but using the fact that `null` has an access type (see IG 8.7.b/T28).

T3. Check that the type of a string literal is determined solely from the context but using the fact that it is a one-dimensional array of character type (see IG 8.7.b/T27).

T4. Check that the character literals in a string literal must be directly visible.

Implementation Guideline: Check when the character literals exist but are not visible, and when the character literals do not exist for the component type.

Check that a one component string literal is allowed.

T5. Check that `CONSTRAINT_ERROR` is raised if the lower bound of a null string literal is 'FIRST of the index base type.

Implementation Guideline: Check in each of the contexts in which the lower bound is determined by the index subtype. IG 4.3.2/T5 requires a similar test for positional aggregates.

Check that `CONSTRAINT_ERROR` is not raised when the lower bound of the literal is determined by a constraint, e.g., as in `A(I..J) := ""`.

T6. Check that `CONSTRAINT_ERROR` is raised if a string literal of array type `A` contains a character that does not belong to `A`'s component subtype.

Implementation Guideline: Include a check for a type that has no graphic character in its subtype.

T7. Check that the lower bound of a string literal is determined correctly (see IG 4.3.2/T14).

4.3 Aggregates

Semantic Ramifications

S1. Note that, syntactically, an aggregate is not a form of name. An aggregate, therefore, cannot be used as a prefix to form selected components, indexed components, slices, or attributes.

S2. A vacuous **others** choice (i.e., a choice that specifies no components) is permissible in an array aggregate, though not in a record aggregate (RM 4.3.1/1):

```
type A is array (1 .. 10) of INTEGER;
...
... A' (1 .. 10 => 0, others => 1) ... -- vacuous others choice
```

RM 4.3/5 says the **others** choice specifies all remaining components, "if any," thereby implying that the choice may specify no components under some circumstances. RM 4.3.1/1 requires that an **others** choice represent at least one component. No similar rule is given in 4.3.2, so an **others** choice in an array aggregate can specify no components.

S3. In an aggregate such as

```
(1..2 => new T) or (A|B => new T)
```

the expression is evaluated once for each component specified by the choice, i.e., 2 times (RM 4.3.1/3 and RM 4.3.2/10). However, $(X|Y \Rightarrow 10)$ can be illegal when $(X \Rightarrow 10, Y \Rightarrow 10)$ is legal:

```
type R is record
  X : INTEGER;
  Y : SHORT_INTEGER;
end record;
REC : R := (X|Y => 10); -- illegal; RM 4.3.1/1
```

There is only one expression, which must have exactly one type. However, since array components must all have the same type, $(1|2 \Rightarrow E)$ is always equivalent to $(1 \Rightarrow E, 2 \Rightarrow E)$.

S4. The rule that named association must be used for an aggregate with only one component eliminates the syntactic ambiguity between such an aggregate and a parenthesized expression, e.g., (9) can never be parsed as an aggregate.

S5. RM 4.3/4 alone suggests that the following aggregate is legal:

```
(2, 4, 6, 3|5 => 0) -- illegal
```

However, this aggregate cannot be a record aggregate (since 3 and 5 are not component identifiers), and RM 4.3.2/3 explicitly states that component associations in an array aggregate (except for an **others** choice) must be either all positional or all named.

S6. Aggregates are subject to the overloading resolution rules discussed in IG 8.7. In particular, the type of an aggregate must be determinable solely from the context in which the aggregate appears, excluding the aggregate itself, but using the fact that this type must be composite and not limited.

```
package P is
  type LP (D : INTEGER := 1) is limited private
```

ND-1589 647

THE ADA (TRADE NAME) COMPILER VALIDATION CAPABILITY
IMPLEMENTERS' GUIDE VERSION 1(U) SOFTECH INC WALTHAM MA
J B GOODENOUGH DEC 86

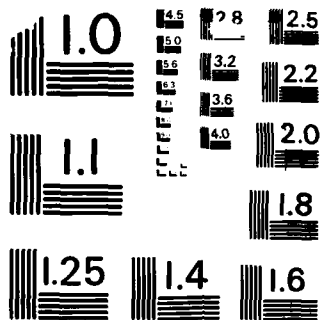
3/9

UNCLASSIFIED

F/G 12/5

三

A 7x14 grid of squares. The top-left corner contains a small cluster of white squares, while the rest of the grid is filled with black squares.



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

```

private
    type LP (D : INTEGER := 1) is ...;
end P;

type REC is
    record
        A : P.LP;
    end record;

procedure PROC (X : LP);
procedure PROC (X : REC);
procedure PROC (X : STRING);
...
PROC((1..3 => 'A'));      -- unambiguous

```

If LP were not a limited type, the PROC call would be ambiguous, since REC would be composite and nonlimited. Note also that LP is not a composite type, even though it has discriminants (RM 3.3/2).

S7. Determination of whether a given aggregate is an array aggregate or a record aggregate is a consequence of the overloading resolution process. If during the top-down phase of the resolution process (see IG 8.7.a/S), a unique type has been determined for the aggregate, then the aggregate is known to be either an array aggregate or a record aggregate. Subsequently, the choices in the aggregate are resolved as component simple names (in the case of a record aggregate) or as simple expressions and discrete ranges (in the case of an array aggregate).

Thus the constituents of a given aggregate have different syntactic and semantic properties depending on whether the aggregate is viewed as a record aggregate or as an array aggregate. Consider the following example:

```

type ENUM is (A, B, C);
type ARR is array (ENUM) of BOOLEAN;
type REC is
    record
        A, B, C => BOOLEAN;
    end record;

```

In REC(A => TRUE, B => FALSE, C => FALSE), A, B, and C are identifiers of a record component (i.e., they are names). In ARR(A => TRUE, B => FALSE, C => FALSE), A, B, and C are expressions (i.e., values).

S8. Visibility of identifiers is different inside array and record aggregates, since the identifier of a record component is directly visible as a choice in a record aggregate of that type (RM 8.3/12). Consider the following example:

```

package P is
    A : constant := 3 ;
    type REC is
        record
            A : INTEGER;
        end record;
    type ARR is array (3..3) of INTEGER;
    procedure PROC (X : REC);
    procedure PROC (X : ARR);
end P;

```

```

package body P is
    ...
begin
    PROC ((A => 1));           -- ambiguous
end P;

procedure PROC (X : P.REC) is ... end PROC;
...
PROC ((A => 1));             -- OK (1)

```

The last aggregate is unambiguously of type P.REC since only one declaration of PROC is directly visible; the aggregate is legal since component A is directly visible as a choice. (Note that (1) would be ambiguous in the presence of use P since in that case there would be more than one possible resolution for PROC.) Also note that the first call would be ambiguous even if it were PROC ((3 => 1)), since the form of the choices cannot be used to resolve the type of the aggregate (RM 4.3/7).

In short, the set of possible types of an aggregate must include all nonlimited composite types in scope at the point of occurrence of the aggregate.

Changes from July 1982

S9. The fact that there are no aggregates for limited composite types or private types with discriminants can be used to resolve the type of an aggregate.

Changes from July 1980

S10. An aggregate is a basic operation, and hence, aggregate notation is available throughout the scope of a composite type.

S11. The type of an aggregate must be determined independently of the form of the choices, the types of the choices, the types of the expressions, the presence or absence of choices (including an others choice), the number of component associations, and whether or not an expression or choice is static.

Legality Rules

These rules apply to both record and array aggregates.

- L1. If both positional and named associations appear in the same aggregate, the positional associations must occur first (RM 4.3/4).
- L2. An aggregate containing only one component association must be given in named notation (RM 4.3/4).
- L3. The choice others, if present, must appear alone and in the last component association (RM 4.3/5).
- L4. The type of an aggregate must be uniquely determined by the context in which it appears, excluding the aggregate itself, but using the fact that the type must be composite and nonlimited (RM 4.3/7).
- L5. An aggregate cannot be written for a limited type (RM 3.6.2/1, RM 3.7.4/1, and RM 7.4.4/11).

Exception Conditions

The exception conditions common to both array and record aggregates are presented here. These exceptions are all associated with the component values specified in an

aggregate. Additional exceptions associated with the bounds of an array aggregate are given in IG 4.3.2/E.

For an aggregate of type T, let E be an expression whose value is associated with a component of type TC. CONSTRAINT_ERROR is raised if E's value does not belong to TC's subtype (RM 4.3.1/3 and RM 4.3.2/11). Depending on TC's base type and E's value, CONSTRAINT_ERROR is raised specifically under the following conditions.

- E1. If TC is a scalar type and the component is not a discriminant of T, CONSTRAINT_ERROR is raised if the value of E lies outside the range specified for TC (RM 4.3.1/3, RM 4.3.2/11, RM 3.3/4, and RM 3.5/3).
- E2. If TC is an array type (note: TC must be a constrained array type), CONSTRAINT_ERROR is raised if at least one bound of E is not equal to the corresponding bound specified for TC (RM 4.3.1/3, RM 4.3.2/11, RM 3.3/4, and RM 3.6.1/4).
- E3. If TC is a constrained record, private, or limited private type, CONSTRAINT_ERROR is raised if the values of E's discriminants do not equal the values specified in TC's discriminant constraint (RM 4.3.1/3, RM 4.3.2/11, RM 3.3/4, and RM 3.7.2/4).
- E4. If TC is a constrained access type designating an object of an unconstrained array, record, private, or limited private type, DT, and E is not null, CONSTRAINT_ERROR is raised if (RM 4.3.1/3, RM 4.3.2/11, RM 3.3/4, and RM 3.8/6):
 - any index bound of the object designated by E does not equal the corresponding bound specified for TC (RM 3.6.1/4).
 - any discriminant of the object designated by E does not equal the corresponding value specified for TC (RM 3.7.2/6).

Test Objectives and Design Guidelines

- T1. Check that aggregates cannot be used as prefixes to form selected components, indexed components, slices, or attributes.

Check that aggregates cannot be written for limited array or record types.

Implementation Guideline: Use record and array types that have a task component or a component having a limited type.

- T2. Check that an array or record aggregate cannot contain any of the following:

- an empty list of component associations;
- a single positional association;
- two component associations with the choice others;
- a choice others in any but the last component association;
- a choice others not appearing alone, as in the following example

1|2|3|others => 0

- a positional association following a named association.

- T3. Check that an expression associated with more than one component is evaluated once for each component (see IG 4.3.1/T7, IG 4.3.2/T7, and IG 4.3.2/T8).

- T4. Check that CONSTRAINT_ERROR is raised under the following circumstances, where TC is the subtype of the component for which an expression is provided in the aggregate:

- a value for a (non-discriminant) component of a scalar type is not within the range of TC.

Implementation Guideline: Use integer, enumeration, float, and fixed types, including derived types.

- the bounds specified for a component of an array type do not equal the bounds of TC.

Implementation Guideline: Use both positional and named notation to determine the bounds. Include a case where the bounds depend on discriminants. Use static and non-static bounds (both for TC and for the component in the aggregate). Use null and non-null arrays.

- for a component of a constrained record or private type, the value of a discriminant in the aggregate does not equal the value of the corresponding discriminant of TC.

Implementation Guideline: Include a case where the discriminant depends on a discriminant. Use static and nonstatic discriminants.

- for a component of an unconstrained record or private type, a discriminant value in the aggregate lies outside the range specified for a discriminant of TC.

- for a component of a constrained access type designating an object of an array, a record, or a private type:

- an index bound of the designated object does not equal the bound specified for TC.

- a discriminant of the designated object does not equal a discriminant value specified for TC.

Implementation Guideline: Do not use an allocator in the aggregate.

Implementation Guideline: Use both array and record aggregates, including aggregates within aggregates.

Implementation Guideline: When components have different subtypes and their value is specified by a single component association, check that the subtype checks are performed for each component.

- T5. Check that the following are not used in overloading resolution to resolve the type of an aggregate:

- the number of array dimensions;
- the number of component associations;
- whether the use of an others choice would be illegal;
- the types of the expressions in the associations;
- the values of the expressions in the associations;
- use of mixed positional and named component associations;
- the staticness or nonstaticness of choices in the aggregate;
- the visibility of names used as choices in the aggregate;
- the fact that CONSTRAINT_ERROR will be raised by certain alternative resolutions.

Check that if a composite type is declared in the visible part of a package, any type conversion within the scope of the package is illegal if the conversion operand is an aggregate (see IG 4.6/T2).

4.3.1 Record Aggregates

Semantic Ramifications

S1. Although a single value may be specified for more than one component in a record aggregate, different components receiving this value might have different subtype constraints associated with them, as in the example below.

```

type R is record
  X : INTEGER range 1..100;
  Y : INTEGER range 90..110;
end record;
A : R := (X|Y => 100);    -- OK
B : R := (X|Y => 10);     -- CONSTRAINT_ERROR

```

For each component association, the expression is evaluated and any applicable constraints must be checked.

S2. When two record components have the same array type with different bounds, the bounds are checked separately even when a single expression gives the value for each component:

```

type REC is
  record
    C1 : STRING (1..2);
    C2 : STRING (2..3);
  end record;
V1 : REC := (C1 | C2 => (others => ' '));
V2 : REC := (C1 | C2 => (1..2 => ' '));    -- CONSTRAINT_ERROR

```

The first aggregate is evaluated as though it had been written with the same expression for each component (AI-00244), i.e., (C1 => (others => ' '), C2 => (others => ' ')). No exception is raised in this case; the first array aggregate has bounds 1..2 and the second, 2..3. The second aggregate is equivalent to writing (C1 => (1..2 => ' '), C2 => (1..2 => ' ')). CONSTRAINT_ERROR is raised because the second array aggregate does not belong to C2's subtype -- the bounds are incorrect, and no implicit subtype conversion is applied to the aggregate (RM 4.3.2/11).

S3. If an aggregate is used for a record type with a variant part, the expression given for the discriminant governing the variant part must be static. The requirement that the expression be static, however, is not used for overloading resolution. For example, consider the following:

```

type R1 is
  record
    B : BOOLEAN;
    I : INTEGER;
  end record;

type R2 (B : BOOLEAN) is
  record
    case B is
      when TRUE =>
        I : INTEGER;
      when FALSE =>
        null;
    end case;
  end record;

```

```

procedure P (X : R1);
procedure P (X : R2);
function F return BOOLEAN;

```

```

...
P ((B => F, I => 10));  -- ambiguous, although only legal for R1

```

The aggregate is ambiguous. The fact that it is legal only if considered of type R1 must not be used for overloading resolution.

S4. Although the value (or type) of a component may not be used to resolve the type of an aggregate, once the type has been determined, the value of a discriminant governing a variant part must be used to determine the types of the components belonging to the variant part, as shown in the following example:

```

type R3 (D : BOOLEAN) is
  record
    case D is
      when TRUE =>
        I1 : INTEGER;
      when FALSE =>
        I3 : LONG_INTEGER;
    end case;
  end record;

```

```

X : R3 := (TRUE, 5);  -- 5 is unambiguously of type INTEGER

```

Given that the aggregate is type R3, the value of the discriminant shows that 5 has type INTEGER, i.e., the value of a discriminant is used to resolve the *subtype* of an aggregate as well as the type of an aggregate's *component*. In addition, the value of a discriminant may imply an aggregate is illegal for a particular subtype, but such illegality does not affect overloading resolution. For example:

```

type R4 (D : BOOLEAN) is
  record
    case D is
      when TRUE =>
        null;
      when FALSE =>
        I3 : FLOAT;
    end case;
  end record;
procedure PROC (X : R3);
procedure PROC (X : R4);
...
PROC((TRUE, 1));  -- ambiguous, although only legal for R3

```

S5. In all cases, the type of the aggregate must be determined without considering the internal structure of the aggregate itself (RM 4.3/7). Thus, the various legality considerations and constraint requirements that pertain to aggregates are not considered for overloading resolution, and are not applied until after overloading resolution has determined a unique type for the aggregate.

S6. Although a named association with the choice others may appear in an array aggregate with positional associations for the preceding components, no other form of named association is allowed in an array aggregate that contains at least one positional association. Hence only record aggregates may have both positional and named associations in their full generality.

S7. If a discriminant does not govern a variant part, it need not be specified by a static expression:

```

type R5 (A, B : POSITIVE; C : BOOLEAN) is
  record
    D : STRING(1..A);
    E : R3(C);
  end record;
X : R5 := (F, G, H, (1..F => 'D'), E => (TRUE, 3));

```

The above aggregate is legal. Neither F, G, nor H need be static. Note, however, that **CONSTRAINT_ERROR** must be raised if H /= TRUE, since the subtype of component E is determined by the value of H. Similarly, **CONSTRAINT_ERROR** will be raised if both evaluations of F do not yield the same value. Finally, the value TRUE in the aggregate for E must be given with a static expression since this discriminant does govern a variant part.

S8. A discriminant value can be determined by an others choice:

```

type R6 (A : BOOLEAN) is
  record
    case A is
      when TRUE =>
        B : BOOLEAN;
        C : INTEGER;
      when FALSE =>
        D : INTEGER;
    end case;
  end record;

Y : R6 := (C => 3, others => TRUE);

```

Note that since the discriminant governs a variant part, the discriminant value must be static, and therefore, the expression governed by others must be static.

Approved Interpretations

S9. In a record aggregate, a component association having multiple choices denoting components of the same type is considered equivalent to a sequence of single choice component associations representing the same components (AI-00244).

Changes from July 1982

S10. There are no significant changes.

Changes from July 1980

S11. Expressions in aggregates are evaluated once for each component (not once for each textual occurrence).

S12. If the choice others is given in a record aggregate, it must represent at least one component.

Legality Rules

L1. An aggregate must specify exactly one value for each component of a record subtype (RM 4.3/6), i.e.:

- each choice in a component association must be an identifier denoting a

component of the record subtype, unless the identifier is the reserved word **others** (in particular, choices cannot be specified for components of a nonexistent variant);

- two choices must not be the same identifier;
 - a choice in a component association cannot specify a component whose value is also specified positionally;
 - a value must be specified for every component of a record subtype, and for only those components.
- L2. The type of each expression must be the same as the type of the corresponding component (RM 4.3.1/1).
- L3. A component association with more than one choice or with the single choice, **others**, is only allowed if the denoted components have the same type and the expression has that type (RM 4.3.1/1).
- L4. If the choice **others** is given, it must represent at least one component (RM 4.3.1/1).
- L5. The value specified for a discriminant governing a variant part must be given by a static expression (RM 4.3.1/2).
- L6. Neither a discrete range nor an expression may be used as a choice in a record aggregate (RM 4.3/5).
- L7. All the legality rules applicable to aggregates in general (IG 4.3/L) must be satisfied.

Exception Conditions

See IG 4.3/E.

Test Objectives and Design Guidelines

T1. Check that each of the following is illegal:

- two choices with the same identifier.
- a choice naming a component whose value was given previously by a positional component association.
- a choice that is not the identifier of a component.
Implementation Guideline: The choice should be the identifier of a component belonging to a different subtype of the record, i.e., variant records should be used.
- a component association with more than one choice or with the single choice **others**, where the corresponding components have different types.
Implementation Guideline: The corresponding expression should be chosen so that it could have any of the types in question.
- a component association with the choice **others**, where the **others** choice does not represent at least one component of the record.
- a value is not provided for every component of the record subtype.
Implementation Guideline: Try one case with a variant record type. Use both positional and named aggregates.
- positional associations following a named association (see IG 4.3/T2).
- a range using component names for lower and upper "bounds".

Implementation Guideline: These checks must be repeated for the following cases, where appropriate:

- the aggregate is not overloaded.
- the aggregate is a legal aggregate of some record type R1, but fails the check for some other record type R2, and its type is not uniquely determined by the context. The aggregate must be considered illegal.
- the aggregate is a legal aggregate of some array type A, but fails the check for some record type R, and its type is not uniquely determined by the context. The aggregate must be considered illegal.

- T2. Check that the expression giving the value of a discriminant governing a variant part must be static. Check both a nonoverloaded and an overloaded aggregate.

Implementation Guideline: Include a case where the discriminant value is given by an others choice representing more than one component.

Check that the staticness of the expression giving the value of a discriminant governing a variant part is not used to resolve the type of the aggregate.

- T3. Check that if a discriminant does not govern a variant part, its value can be given by a nonstatic expression.

Implementation Guideline: Use a discriminant not used inside the record, a discriminant used as an array index bound, and a discriminant used in a discriminant constraint.

Check that when a discriminant gives the bounds of an array component, the value assigned to the record has the correct bounds.

Implementation Guideline: Include a check for a generic formal type that has discriminants.

- T4. Check that an aggregate is not considered ambiguous if, without considering the value of the discriminants, it could have more than one subtype of a record type with variants.

- T5. Check that in a record aggregate with a form such as $(X \Rightarrow E, Y \Rightarrow E)$, if E is an overloaded expression, overloading resolution occurs separately for the different occurrences of E (i.e., the aggregate is not equivalent to $(X|Y \Rightarrow E)$ if components X and Y have different types).

- T6. Check that both named and positional notation are permitted within the same record aggregate if all positional associations appear first. (IG 4.3/T2 checks that positional associations cannot follow named associations.)

- T7. Check that an expression associated with more than one record component is evaluated once for each associated component.

Implementation Guideline: Include a case where others is associated with more than one component.

Implementation Guideline: Use expressions with allocators, function calls, and function calls combined with operators. Include a case where the expression is an aggregate and the components have different bounds.

- T8. Check that the value of a discriminant is used to resolve the type of a component that depends on the discriminant.

Implementation Guideline: Include a case where the value of a discriminant is determined by an others choice and a case where the others choice must be static.

4.3.2 Array Aggregates

Semantic Ramifications

- s1. The type of an array aggregate must be determinable independent of the names used in choices within the aggregate; the types of the expressions within the aggregate; the forms of the choices (e.g., the use of 1..2 as a choice does not that imply the aggregate is an array aggregate); the use of both positional and named associations in the same aggregate (which is legal only for record aggregates); or the fact that a choice is nonstatic (which may be illegal for an array aggregate) (RM 4.3/7).

S2. The constraints associated with either the component type or the index type must not be used in overloading resolution. In particular, the number of components must not be used:

```
type S3 is new STRING (1..3);
type S5 is new STRING (1..5);
procedure P (X : S3);
procedure P (X : S5);
...
P("ABCDE");      -- ambiguous
```

S3. The requirement that choices be static must be checked *after* overloading resolution, rather than being used for overloading resolution. For example:

```
type MINT is new INTEGER;
function "abs"(X : MINT) return MINT;
type A1 is array (INTEGER range 1..10) of INTEGER;
type A2 is array (MINT range 1..10) of INTEGER;

... (1..abs(10) => 0, others => 1)      -- ambiguous
```

It is irrelevant that the aggregate can only legally have type A1 (since `abs(10)` is nonstatic for type A2, and a nonstatic choice is not allowed together with an `others` choice).

Similarly, if we add the following declarations:

```
type R is
  record
    X, Y : INTEGER;
  end record;

X, Y : MINT := 1;
```

then the aggregate `(X => 0, Y => 1)` has type R or A2. The fact that the aggregate is illegal if it has type A2 is irrelevant when resolving its type.

S4. The fact that a mixture of positional and named notation is forbidden for array aggregates cannot be used in overloading resolution:

```
type R is
  record
    A, B: INTEGER;
  end record;

type E is (A, B);

type A1 is array (E) of INTEGER;

... (0, B => 0)      -- ambiguous
```

It is irrelevant that the aggregate is structurally illegal as an array aggregate.

S5. An array value (and particularly an aggregate) may have an anonymous type introduced by an object declaration. Although subprograms and operators with formal parameters of such a type cannot be declared explicitly (because the type cannot be named), the implicit declaration of operators for the anonymous type can cause ambiguities:

```

type A is array (0..1) of INTEGER;
X : array (0..1) of INTEGER;
Y : A;
...
if (1, 2) = (3, 5) ...      -- ambiguous

```

Three equality operators are visible for composite types (equality for STRING, A, and the anonymous array type), and there is insufficient contextual information to determine which operator should be chosen; hence the type of the aggregates (1, 2) and (3, 5) cannot be uniquely determined.

s6. String literals are not aggregates according to the syntax, but rather are expressions. Therefore,

- a string literal must be enclosed in parentheses in an allocator and type qualification, where an aggregate would not need extra parentheses:

```

new STRING "ABCD"      -- illegal
STRING' "ABCD"         -- illegal

```

- a one-component string literal is allowed:

```

"A"                    -- legal
('A')                  -- illegal aggregate

```

- a null string literal is allowed

```

""                      -- legal
()                      -- illegal null aggregate

```

s7. An array aggregate containing an **others** choice is allowed after := only if the target object has a constrained subtype. The target object can fail to be constrained in two cases:

- the declaration of a constant object, e.g.:

```

X : constant STRING := (others => ' ');      -- illegal others

```

- the declaration of a formal parameter of a subprogram, an entry, or a generic unit, e.g.:

```

procedure P (X : STRING := (others => ' ')); -- illegal others

```

s8. The bounds of an array aggregate are computed only after the type of the aggregate has been determined from the context. If an **others** choice is used in the aggregate, however, the bounds of the aggregate must be determined from the context as well. For this reason, the RM restricts when an **others** choice may be used in an array aggregate (RM 4.3.2/4-8). In particular, an array aggregate with an **others** choice is not permitted in any context that defines an unconstrained subtype for the aggregate. In addition, if the aggregate contains an **others** choice and is used in a context where "sliding" of bounds occurs (see below), no other named associations are permitted.

Since apart from an optional **others** choice, component associations in an array aggregate must be either all named or all positional, determination of the bounds reduces to three cases:

- Named or positional associations plus an **others** choice (or an **others** choice alone):

In the presence of an **others** choice, the bounds of the aggregate are always determined by the applicable index constraint, since in all contexts where an

others choice is permitted, the context determines a unique constrained array subtype for the array aggregate.

- Named associations without an **others** choice:

The bounds are determined by the smallest and the largest choices given. The bounds of a null array aggregate are always given by a single named association. The RM does not say, however, that the largest value in this association is the value of the upper bound, since for a null array, the largest value is the value of the lower bound and the smallest value is the value of the upper bound. Note also that in every legal array aggregate with more than one choice, each choice must be static (RM 4.3.2/3), and therefore, the bounds can be determined at compile time for such aggregates.

- Positional associations without an **others** choice:

The lower bound is determined by the applicable index constraint if the aggregate appears in a context where an **others** choice is permitted; otherwise, the lower bound is defined as S'FIRST where S is the index subtype. In either case, the upper bound is determined by the number of components.

The rules for positional associations without an **others** choice apply also to determination of the bounds of string literals (see RM 4.2/3 and below).

S9. Array assignment and generic instantiation with an array value implicitly convert the array value to the subtype of the target. This conversion adjusts the bounds of the aggregate to match those of the target variable (this is sometimes called "sliding"):

```
type R is
  record
    A : STRING (1..10) := (2..11 => 'A');  -- will slide
  end record;
X : STRING(1..3) := (2 => 'A', 3..4 => 'B');  -- will slide;
...                                           -- X(2) = 'B'
X := (2|3|4 => '0');                        -- will slide
```

S10. Named associations are not allowed together with an **others** choice in contexts where sliding of the bounds occurs:

```
type A is array (POSITIVE range 1..4) of INTEGER;
W : A := (others => 0);                      -- legal
X : A := (1, 2, others => 3);                 -- legal with positional but
Y : A := (1 => 1, others => 0);               -- illegal with named associations
Z : A := A'(1 => 1, others => 0);             -- legal
```

This rule derives from consideration of examples like the following:

```
Z(3..4) := (2 => 0, others => 3);  -- illegal
```

What should the lower bound of the aggregate be: 1, the lower bound of the index subtype, or 3, the lower bound of the applicable index constraint? If the lower bound of the applicable index constraint is chosen, then the explicit choice, 2, cannot be accepted, and presumably, CONSTRAINT_ERROR should be raised. Choosing the lower bound of the index subtype, i.e., 1, would be a different rule from that used for positional aggregates in this context. By making such usage illegal, a programmer is required to say explicitly which of these possible interpretations is intended.

S11. When a null string literal is used in a context where sliding occurs, the lower bound of the

literal is determined by the applicable index constraint rather than by 'FIRST of the index subtype:

```

type ENUM is (A, B, C);
type S is array (ENUM range <>) of CHARACTER;
S1 : S(C..A) := "";           -- no exception
S2 : S(C..B) := (others => 'A'); -- no exception
S3 : BOOLEAN := "" = S'("AB"); -- CONSTRAINT_ERROR

```

CONSTRAINT_ERROR is raised by the null string literal in the declaration of S3 since its type is S, there is no applicable index constraint, and the index subtype's lower bound is ENUM'FIRST. Since there is no predecessor for ENUM'FIRST (RM 4.2/3), CONSTRAINT_ERROR is raised. CONSTRAINT_ERROR is not raised in the declaration of S1 or S2, however, since the lower bound of "" and (others => 'A') is determined by the index constraints (C..A) and (C..B), respectively, i.e., the lower bound is C. (Although the upper bound of the string literal in S1's declaration is B (see RM 4.2/3), there is no way to check on this since the effect of the assignment to S1 is to change the bounds to C..A (RM 5.2.1/1 and RM 4.6/11).) Note that the subtype conversion implicit in the initialization of S1 and S2 cannot be written explicitly since a string literal is not allowed as the operand of a conversion (RM 4.6/3), and type conversion is not one of the contexts in which an aggregate with an others choice is permitted.

S12. If an array aggregate has a null range, it cannot also have an others choice, since static null ranges are required to be the sole choice of a single component association, and nonstatic choices (including nonstatic null ranges) are also required to be the sole choice of a single component association.

S13. If an others choice has no corresponding components (RM 4.3/5), the expression associated with the choice is not evaluated (RM 4.3.2/10). Nonetheless, the expression must have the type of the array component.

S14. If an others choice appears in an aggregate and more than one component association is also present, the corresponding index subtype and discrete range must be static (AI-00310):

```

subtype SMALL is INTEGER range P..Q;           -- P, Q nonstatic

type B1 is array (SMALL range <>) of INTEGER;
X1 : B1(1..5) := (1, others => 2);              -- illegal

type B2 is array (SMALL range <>, INTEGER range <>) of INTEGER;
X2 : B2(1..5, 1..5) := (1..2 => (3, others => 4), -- legal others
                      others => (1..5 => 0));    -- illegal others

```

SMALL is a nonstatic subtype so B1 has a nonstatic index subtype. Consequently, although the discrete range in X1's declaration is static, the others choice is not allowed. In B2's declaration, the first dimension has a nonstatic index subtype; the second index subtype is static. Consequently, the first others choice is allowed (the corresponding index subtype and discrete range are both static), and the second is illegal (the corresponding index subtype is nonstatic).

S15. An others choice in an aggregate can sometimes be associated with both a static and a nonstatic index constraint, in which case, the enclosing aggregate can be illegal:

```

type REC is
  record
    ST : STRING (1..3);
    NS : STRING (1..N);           -- N is nonstatic
  end record;

```

```
OBJ : REC := (ST | NS => (others => ' ')); -- illegal
```

The aggregate is equivalent to:

```
(ST => (others => ' '), -- legal
 NS => (others => ' ')) -- illegal others
```

Since the applicable discrete range for the second `others` choice is not static, the aggregate is illegal (AI-00244).

S16. A generic formal discrete type is not static within the generic unit, but it can denote a static type in an instance. Hence, an `others` choice can be illegal for an array type that is declared in a generic template, but legal for the corresponding type in an instance (AI-00409):

```
generic
  type T is range <>;
package SET_OF is
  type SET is array (T) of BOOLEAN;
  X : SET := SET'(1 => TRUE, others => FALSE); -- illegal
end SET_OF;

subtype SMALL is INTEGER range 1..10;
package SET_OF_SMALL is new SET_OF (SMALL);
subtype SMALL_SET is SET_OF_SMALL.SET;

SET : SMALL_SET := SMALL_SET'(1 => TRUE, others =>); -- legal
```

The index subtype for `SET_OF.SET` is not static within the generic unit since `T` is a generic formal type. But in the instance, `SET_OF_SMALL`, `T` denotes `SMALL`, and `SMALL` is a static type. Hence, `SMALL_SET` has a static index constraint and the use of `others` in the initialization of `SET` is legal (AI-00409).

S17. RM 4.3.2/2 allows a string literal "In a multidimensional aggregate at the place of a one-dimensional array of character type." This rule allows the use of string literal notation even though no one-dimensional array of character type has been declared:

```
type A2 is array (1..2, POSITIVE range 2..3) of CHARACTER;
X1 : A2 := (1..2 => "EF"); -- legal
```

The declaration of `A2` is *not* equivalent to:

```
type anon is array (POSITIVE range 2..3) of CHARACTER;
type two_dim is array (1..2) of anon;
```

No one-dimensional array is implicitly declared, since even after `A2` is declared, the following expression is unambiguous:

```
"AB" = "AB" -- unambiguous
```

It is unambiguous because only one equality operation is visible for a one-dimensional array of a character type, namely, the operation declared in `STANDARD` for the type `STRING`. (Note: `('A', 'B') = ('A', 'B')` would be ambiguous because two equality operators for composite types are visible; hence, the type of the aggregate `('A', 'B')` cannot be determined from the context of its use.)

S18. Although RM 4.3.2/2 allows a string literal to be used in a multidimensional aggregate, it does not allow an expression that delivers a value of type `STRING` to be used, e.g.:

```
X1A : A2 := (1..2 => "E" & "F"); -- illegal
```

"E" & "F" is an expression, not a literal, and so is not allowed as a subaggregate.

S19. There are two kinds of array aggregates: multidimensional aggregates, whose form is specified in RM 4.3.2/2, and one-dimensional aggregates that are not subaggregates of multidimensional aggregates. There are several significant differences between these kinds of aggregates. For example, consider the use of an **others** choice together with named associations:

```
type AS is array (1..2) of STRING(2..3);
X2 : A2 := (1..2 => (2 => 'E', others => 'F')); -- illegal
Z2 : AS := (1..2 => (2 => 'E', others => 'F')); -- legal
```

For X2, the outer aggregate has a multidimensional array type; hence, the inner aggregate is a subaggregate, and the legality of the **others** choice depends on the context in which the multidimensional aggregate is used (RM 4.3.2/6 and AI-00177). In this case, the context does not allow an **others** choice together with named associations (RM 4.3.2/6). For Z2, the outer aggregate has a one-dimensional array type; the inner aggregate is therefore not a subaggregate, but instead, is an expression of a component association. An **others** choice is allowed in this context (RM 4.3.2/8); moreover, since the restrictions on named associations (RM 4.3.2/6) don't apply in this context, the named associations are allowed as well.

S20. The syntactic form of a multidimensional array aggregate is fully specified in RM 4.3.2/2. In particular, extra parentheses are not allowed to enclose a subaggregate of a multidimensional aggregate. Hence:

```
X3 : A2 := (1..2 => ("EF")); -- illegal parentheses
Z3 : AS := (1..2 => ("EF")); -- legal
```

For Z3, the extra parentheses are allowed since ("EF") is not a subaggregate.

S21. Although all choices of an aggregate must, in principle, be evaluated, and although all expressions associated with non-vacuous choices must be evaluated, the RM does not specify whether all choices and expressions must be evaluated before the checks in RM 4.3.2/11 are made. RM 4.3.2/10 simply specifies that no expression in a component association can be evaluated until all choices have been evaluated; RM 4.3.2/11 simply specifies what checks must be made when values are known. In the absence of specific wording stating that the checks can only be made after all choices are evaluated, an implementation is free to make the checks as soon as possible. For example:

```
AS' (F..G => (H..I => 'X')) -- (1); one-dimensional
```

If F..G is not a null range, an implementation is allowed to check that the values of F and G belong to the index subtype prior to evaluating the inner aggregate. However, since the index subtype checks need not be performed before evaluating the expressions of an aggregate, it is also permissible to evaluate H and I prior to checking the F and G values, and if H..I is not a null range, the expression 'X' can be evaluated before checking that F, G, H, and I belong to the index subtype. If F..G is a null range, then H, I, and 'X' must not be evaluated (RM 4.3.2/10).

S22. The evaluation of choices in a multidimensional aggregate proceeds differently from the evaluation for a one-dimensional aggregate whose component type is an array:

```
A2' (F..G => (H..I => 'X')) -- (2); multidimensional
```

If F..G is a non-null range and F or G does not belong to the index subtype, CONSTRAINT_ERROR can be raised before evaluating H or I, and must be raised even if H..I is a null range (AI-00313). If F..G is a null range, then H..I must be evaluated to see whether it also is a null range. If it is not, CONSTRAINT_ERROR must be raised if H or I does not belong to the index subtype. Since the order of evaluation of choices is not defined for a multidimensional array, the range H..I could be evaluated and checked before evaluating F or G.

If F..G and H..I are both non-null ranges and no exceptions are raised, F, G, H, and I are evaluated just once (RM 4.3.2/10). In the one-dimensional case, if F..G and H..I are both non-null ranges and no exceptions are raised, then H and I are each evaluated G-F+1 times, since the expression in the aggregate must be evaluated once for each component.

In short, the semantic differences between the evaluation of choices and expressions in one-dimensional and multidimensional aggregates can be summarized as follows:

- if F..G is a null range, then
 1. H must be evaluated (in the multidimensional case);
 2. H must not be evaluated (in the one-dimensional case);
- if F..G is not a null range and F does not belong to the index subtype, CONSTRAINT_ERROR can be raised before or after evaluating H, I, or 'X', in either the multidimensional or one-dimensional case.
- if F..G is not null and F and G belong to the index subtype,
 1. H and I must be evaluated once (if neither raises an exception) (the multidimensional case);
 2. H and I must be evaluated G-F+1 times, i.e., once for each component of subtype AS (the one-dimensional case).

The above conclusions are not changed even if F..G is replaced with a static range.

S23. Now consider:

```

A2' (1 => (H..I => 'X'),
     2 => (H..I => 'Y'))      -- (3)
AS' (1 => (H..I => 'X'),
     2 => (H..I => 'Y'))      -- (4)

```

For the multidimensional aggregate in (3), H and I are evaluated once for each textual occurrence as a choice (unless an exception is raised before all the evaluations are done); an exception could be raised for any of the following reasons:

- the evaluation of H or I raises an exception.
- a range, H..I, is non-null, H or I does not belong to the index subtype, and this check is made before evaluating all occurrences of H and I.
- a check that all the subaggregates have the same bounds is done before all occurrences of H and I are evaluated.

S24. For the following multidimensional array:

```

(1..2 => (F..G => F1),
 3..4 => (H..I => F2))

```

CONSTRAINT_ERROR can be raised as soon as:

- F and H have been evaluated and it has been determined that F /= H;
- F and G have been evaluated, F..G does not define a null range, and the value of either F or G does not belong to the index subtype.

S25. Just as all choices do not have to be evaluated before making certain checks, so all expressions do not have to be evaluated before checking to see if an expression's value

belongs to a component subtype. **CONSTRAINT_ERROR** can be raised as soon as a violation of a component subtype constraint is detected. For example, if F1 above does not satisfy the component constraint, then F2 need not be evaluated.

S26. Note that the reason for not evaluating certain choices or expressions in an aggregate is independent of the optimization rules given in RM 11.6. The RM leaves an implementation free to perform the checks before all the choices or expressions are evaluated. If the consequence of performing a check is to raise an exception, then an additional consequence is that certain expressions or choices may not have yet been evaluated.

S27. A multidimensional aggregate can specify a null array even if no choices in the aggregate are null:

```
type S2 is array (POSITIVE range <>,
                  POSITIVE range <>) of CHARACTER;
X6 : S2(4..3, 1..2) := (F..G => "");    -- no exception
```

No exception is raised for the above aggregate unless F..G is non-null and the value of F or G does not belong to the index subtype, **POSITIVE**. The lower bound of the string literal is 1, since the second discrete range of (4..3, 1..2) determines the lower bound. Hence, **CONSTRAINT_ERROR** will be raised in the following case:

```
X7 : S2(4..3, INTEGER'FIRST..-1) := (F..G => "");
```

The lower bound of "" is **INTEGER'FIRST**, which has no predecessor, and so **CONSTRAINT_ERROR** is raised (RM 4.2/3) before the implicit subtype conversion is applied.

S28. For multidimensional aggregates, the bounds of each subaggregate are determined independently:

```
X4 : A2 := (1 => "AB",
            2 => (3 => 'D', 4 => 'E'));
```

CONSTRAINT_ERROR will be raised because the bounds of "AB" will be 2..3, and these bounds do not equal the bounds of the second subaggregate. Note that **CONSTRAINT_ERROR** cannot be raised because the bounds of the subaggregate in the second component association are 3..4 instead of the bounds of A2's second dimension. The following aggregate would raise no exception:

```
X5 : A2 := (1 => (3..4 => 'A'),
            2 => (3..4 => 'B'));
```

No exception is raised because the index subtype for A2's second dimension is **POSITIVE**, not 2..3 (see RM 3.6/5), and the bounds of the multidimensional aggregate are not required to match the bounds of A2's subtype. Moreover, the subtype conversion associated with assignment adjusts the bounds of the second dimension. The same aggregate would give rise to **CONSTRAINT_ERROR** in the absence of the subtype conversion, e.g., if the aggregate were used as an actual parameter when the formal parameter's subtype was A2.

S29. Consider the same aggregate assigned to a variable of type AS:

```
Z5 : AS := (1 => (3..4 => 'A')
            2 => (3..4 => 'B'));
```

CONSTRAINT_ERROR will be raised because the subtype of the inner aggregate must satisfy the subtype constraint for AS's component (RM 4.3.2/11), and AS's component subtype is **STRING(2..3)**.

S30. If a positional aggregate (or string literal; see RM 4.2/3) is enclosed in parentheses, the

bounds of the aggregate (or literal) are determined by the index subtype since the aggregate no longer appears in one of the contexts specified by RM 4.3.2(a-c), e.g.:

```
X6 : AS := (1..2 => ("EF"));      -- CONSTRAINT_ERROR
```

The bounds of "EF" are 1..2, but the AS component subtype requires bounds 2..3. Hence, CONSTRAINT_ERROR will be raised. CONSTRAINT_ERROR would not be raised if the parentheses enclosing "EF" were omitted, since the bounds of the string literal would then be determined by the applicable index constraint, 2..3.

S31. Even for null array aggregates, subaggregates must have the same bounds. For example:

```
(3..4 => (1..0 => 0),
 5..6 => (2..1 => 0))      -- CONSTRAINT_ERROR

(1..0 => (3..4 => (1..2 => 0)
        (5..6 => (7..8 => 0))))  -- CONSTRAINT_ERROR
```

In the first example, the null choices do not have the same bounds; in the second example, the non-null choices, 1..2 and 7..8, have different bounds.

S32. AI-00019 specifies that the bounds of a positional aggregate must belong to the corresponding index subtype. In particular, consider:

```
type ENUM is (A, B, C, D);
subtype SMALL is ENUM range A..C;
type ARR is array (SMALL range <>) of INTEGER;
function F (CHOICE : BOOLEAN) return ARR is
begin
    return (1, 2, 3, 4);      -- CONSTRAINT_ERROR
end F;
```

The lower bound of (1, 2, 3, 4) is A and the upper bound does not belong to the index subtype, SMALL, so CONSTRAINT_ERROR is raised when the aggregate is evaluated.

Approved Interpretations

S33. An **others** choice is static if the corresponding index subtype is static and the corresponding index bounds are specified with a static discrete range (AI-00310).

S34. If an index subtype has a generic formal type (or a type derived directly or indirectly from a generic formal type), the index subtype is not static within the generic unit, so aggregates can have only a single component association with a single choice (AI-00190).

S35. For a generic instantiation, if an actual generic parameter is a static subtype, then every use of the corresponding formal parameter within the instance is considered to denote a static subtype, even though the formal parameter does not denote a static subtype in the generic template (AI-00409).

S36. If an aggregate containing an **others** choice is the expression of the component association of an enclosing array aggregate, and the aggregate containing the **others** choice is a subaggregate, the **others** choice is allowed (with or without additional named associations) if and only if the enclosing multidimensional aggregate is in a context that allows an **others** choice (with or without additional named associations). If the aggregate with the **others** choice is not a subaggregate, the **others** choice is allowed (with or without additional named associations) (AI-00177).

S37. For positional aggregates, a check is made that the index bounds belong to the corresponding index subtype; CONSTRAINT_ERROR is raised if this check fails (AI-00019).

S38. For the evaluation of a non-null dimension of a multidimensional aggregate, a check is made that the index values defined by choices belong to the corresponding index subtype. The exception `CONSTRAINT_ERROR` is raised if this check fails (AI-00313).

Changes from July 1982

S39. Named associations are not allowed in an aggregate with an `others` choice that appears as an actual generic parameter.

S40. If an object is declared with an anonymous array type, it can be initialized with an aggregate that has an `others` choice.

S41. For a multidimensional aggregate, a check is made that all (n-1)-dimensional subaggregates have the same bounds.

Changes from July 1980

S42. A null range must not be given if there is more than one component association or more than one choice in the aggregate. (Hence more than one null range is forbidden.)

S43. An aggregate with an `others` choice (but no other named associations) is allowed after `:=` in an assignment statement, constrained formal parameter declaration, constrained object declaration, or record component declaration.

S44. An aggregate with an `others` choice is allowed as a generic actual parameter.

S45. Rules for evaluating an aggregate's choices and expressions are given. In particular, an expression is evaluated once for each choice specified in a named association, and the evaluation order of expressions is explicitly not defined by the language.

S46. `CONSTRAINT_ERROR` is raised if the value of a subcomponent does not belong to the subtype of the subcomponent.

S47. In a multidimensional array aggregate, the one-dimensional subaggregates must have the form of either string literals or aggregates; arbitrary expressions yielding suitable one-dimensional array values are not allowed. Similarly, (n-1)-dimensional subaggregates must be written as aggregates when $n > 2$ (i.e., no other form of expression is allowed).

Legality Rules

- L1. The type of each choice must be the type of the corresponding index (RM 4.3.2/2).
- L2. The type of an expression specifying an array component value must be the same as the type of the component (RM 4.3.2/1).
- L3. If an aggregate has an n-dimensional array type where $n > 1$, the aggregate must be written as a one-dimensional aggregate in which each expression is written as an (n-1)-dimensional subaggregate, and similarly, an n-dimensional subaggregate (for $n > 1$) must be written as a one-dimensional aggregate in which each expression is an (n-1)-dimensional subaggregate. However, when $n = 2$ and the array's component type is a character type, an expression in the two-dimensional subaggregate may have the form of a string literal (RM 4.3.2/2).
- L4. An array aggregate must not contain a mixture of positional and named component associations unless the only named component association is the last component association, and this association has the single choice `others` (RM 4.3.2/3).
- L5. When more than one choice is given, every choice must be static and non-null, i.e., for choices of the form `E`, `L..R`, `ST`, and `ST range L..R`, `L`, `R`, and `E` must be static expressions, `ST` must denote a static subtype, and the discrete ranges `L..R`, `ST`, and `ST range L..R` must not be null (RM 4.3.2/3).

- L6. If **others** is a choice in an aggregate containing more than one component association, the corresponding index subtype and discrete range must both be static (RM 4.3.2/3 and AI-00310).
- L7. An array aggregate (one-dimensional or multidimensional) with the choice **others** can only appear:
- as an actual parameter corresponding to a constrained formal parameter of a subprogram or entry (RM 4.3.2/5-6);
 - as an actual generic parameter corresponding to a constrained generic formal parameter, and where no other named associations are used in the aggregate (RM 4.3.2/5);
 - in a return statement in a function whose return type is constrained (RM 4.3.2/5-6);
 - as the expression following **:= in:** in an assignment statement, a record component declaration, the declaration of a formal parameter of a subprogram, entry, or generic unit when the formal parameter is constrained, or the declaration of a variable or constant having a constrained subtype indication; all component associations must be positional, except for the association having the choice **others** (RM 4.3.2/5-6);
 - in a qualified expression (or allocator), where the type mark denotes a constrained array subtype (RM 4.3.2/7);
 - as the expression of an enclosing record or array aggregate if the aggregate with the choice **others** (RM 4.3.2/8 and AI-00177):
 - is not a subaggregate (named associations are also allowed);
 - is a subaggregate of a multidimensional aggregate that occurs in a context that allows an **others** choice (named associations are allowed in addition if the multidimensional aggregate appears in a context that allows an **others** choice with such named associations).
- L8. For an array aggregate using named associations without an **others** choice, the set of values covered by the choices must be complete, i.e., if the minimum choice value is L and the maximum is R, every value in the range L .. R must be represented exactly once in the set of choices (there must be no omissions and no duplicates) (RM 4.3/6).
- L9. A choice must be an expression or a discrete range (RM 4.3/5).
- L10. All the restrictions of IG 4.3/L must be satisfied.

Exception Conditions

Note: We list here only those conditions under which the type of an array aggregate alone is sufficient to determine that **CONSTRAINT_ERROR** should be raised. When such an aggregate is used as an initialization, assigned value, actual parameter, return value, or operand of a qualification, additional conditions determine whether an exception must be raised. These conditions are specified explicitly in each of these contexts (see IG 3.6/E, IG 3.7.2/E2, IG 4.3/E, IG 4.7/E, IG 5.2.1/E, IG 5.8/E, and IG 6.4.1/E).

The conditions listed in IG 4.3/E are in addition to the following:

- E1. For a multidimensional aggregate, **CONSTRAINT_ERROR** is raised if the subaggregates for a given dimension do not all have the same bounds (RM 4.3.2/11).

- E2. For a multidimensional or one-dimensional aggregate, **CONSTRAINT_ERROR** is raised if the lower or upper bound for at least one non-null discrete range does not satisfy the range constraint of the associated index subtype (RM 4.3.2/11 and AI-00313).
- E3. **CONSTRAINT_ERROR** is raised for a positional aggregate if the upper bound does not belong to the index subtype (AI-00019).

Test Objectives and Design Guidelines

T1. Check that:

- an array aggregate must not contain a positional component association preceding a named association that does not have the choice **others**.
- a choice must not be a nonstatic expression or a static null discrete range in an aggregate with more than one component association or more than one choice.

Implementation Guideline: Try choices of the form E, L..R, ST, ST range L..R, and A'RANGE (where A is an array object or a constrained array type with static bounds and a static index subtype.)

Implementation Guideline: Include cases such as (F..G => 0), **others** => 1) and (2..1 => 0, **others** => 1) when the index subtype is static.

Implementation Guideline: In some cases, the index subtype should be a generic formal discrete type, or a type derived directly or indirectly from a formal type.

- if an aggregate has more than one choice or component association and one choice is **others**, the corresponding index subtype and discrete range (AI-00310) must be static.

Implementation Guideline: Include a use of a null static range and a vacuous **others** choice.

Implementation Guideline: Check where the component associations are both positional and named.

Implementation Guideline: Include a case where the subtype and index bounds for an index are static, but another dimension has either a nonstatic index subtype (with static index bounds) or nonstatic index bounds.

- for a non-null dimension of an aggregate, no index value between the lower and upper bound of the aggregate can be left uncovered by the set of choice values.

Implementation Guideline: Include a null multidimensional aggregate with one non-null dimension. e.g.,

```
(1..2 => (2..1 => 1),
-- 3 omitted
4..5 => (2..1 => 2))
```

- an index value must not be represented more than once in the set of choices.

Implementation Guideline: Check for ranges that overlap, for duplicate choice values, and for an overlap between a choice value and a range.

- the type of a choice must be the same as the corresponding index type.

- the type of the expression specifying an array component value must be the same as the type of the array component.

Implementation Guideline: Include a case where the expression associated with a vacuous **others** choice is not the correct type.

- the innermost subaggregate of a multidimensional aggregate cannot be enclosed in parentheses.

Implementation Guideline: Check for both one-dimensional and multidimensional aggregates.

T2. Check that an array aggregate with an **others** choice is illegal:

- as an initial value in a constant declaration where the subtype indication specifies an unconstrained array type.
- in the declaration of an object or formal parameter (of a subprogram, entry, or generic unit) when the object or parameter has a constrained array subtype and additional named associations are used in the aggregate.
- in the declaration of a subprogram, entry, or generic formal parameter that has an unconstrained array type.
- in an assignment statement when additional named associations are used in the aggregate.
- as the operand of a predefined operator when the context specifies a constrained array subtype (e.g., `F(not(others => TRUE))`), where `F`'s formal parameter is constrained).

Implementation Guideline: Check for `not`, `and`, `or`, `xor`, `"="`, `"/="`, `"&"`, `">"`, `">="`, `"<"`, and `"<="`.

- as the expression in a membership test when the type mark denotes a constrained or unconstrained array type.
- in a qualified expression when the type mark denotes an unconstrained array type.
- as an actual `in` parameter corresponding to an unconstrained formal parameter of a subprogram, entry, or generic unit.
- as an actual `in` parameter corresponding to a constrained formal parameter of a generic unit, when additional named associations are present.
- in a return statement in a function returning a result of an unconstrained array type.
- when the corresponding index subtype or discrete range is nonstatic and more than one component association is present (see T1).

Implementation Guideline: Include cases where the aggregate appears as a subaggregate.

Implementation Guideline: When possible, for every illegal case with a multidimensional aggregate, create a legal aggregate having the same form, but for an array of array type.

T3. Check that a parenthesized array aggregate with an **others** choice is illegal:

- as an actual `in` parameter of a subprogram call, an entry call, or a generic instantiation, when the formal parameter is constrained.
- as the result expression of a function when the result type is constrained.
- as the initialization expression of a constrained constant or variable object declaration, constrained formal parameter (of a subprogram, entry or generic unit), or record component declaration.

Implementation Guideline: Include initialization of a variable having an anonymous array type.

- as the expression in an assignment statement.
- as the operand in a qualified expression when the type mark denotes a constrained array subtype.
- as an expression specifying the value of an array or record component.

Implementation Guideline: Use parenthesized expressions in both one-dimensional and multidimensional aggregates.

Implementation Guideline: Try both multidimensional and one-dimensional parenthesized aggregates.

Implementation Guideline: Use both positional and named associations when possible, i.e., except in the generic instantiation and after :=.

- T4. Check that an aggregate with an **others** choice can appear in the following contexts, and that the bounds of the aggregate are determined correctly (e.g., no CONSTRAINT_ERROR is raised and the components covered by the **others** choice have the correct values):

- an actual parameter of a subprogram call, an entry call, or a generic instantiation, when the formal parameter is constrained.
- the result expression of a function when the result type of the function is a constrained array type.
- the initialization expression of a constrained constant or variable object declaration, constrained formal parameter (of a subprogram, entry, or generic unit), or record component declaration.

Implementation Guideline: Include initialization of a variable having an anonymous array type.

- as the expression in an assignment statement.
- as the operand in a qualified expression when the type mark denotes a constrained array subtype.
- as an expression specifying the value of an array or record component.

Implementation Guideline: Use both one-dimensional and multidimensional aggregates for which a component value is specified with an **others** choice.

Implementation Guideline: Try both multidimensional and one-dimensional aggregates with **others** choices.

Implementation Guideline: Use both positional and named associations, except for the generic instantiation and after :=.

Implementation Guideline: Include cases where the **others** association is the only association and the corresponding index subtype or discrete range is nonstatic.

Implementation Guideline: Include cases where more than one component association is present.

Implementation Guideline: Include a multidimensional array case in which some dimensions have static index subtypes and some do not, and check that an **others** choice is allowed when the index subtype is static (see also T1).

Implementation Guideline: All the above contexts should be checked for both null and non-null aggregates. In particular, for null aggregates, **FIRST** of the index subtype should have no predecessor in some cases.

- T5. Check that the bounds of a positional aggregate are determined correctly. In particular, check that the lower bound is given by:

- **FIRST** of the index subtype when the positional aggregate is used as:
 - an actual parameter in a subprogram or entry call, and the formal parameter is unconstrained;
 - an actual parameter in a generic instantiation, and the formal parameter is unconstrained;
 - the return expression in a function whose return type is unconstrained;
 - the initialization expression of a constant whose type mark denotes an unconstrained array;
 - the left or right operand of "&", the relational operators, and the equality operators;

- an expression enclosed in parentheses when the value of the expression is the value of a record component or array component.
- the lower bound of the applicable index constraint when the positional aggregate is used as:
 - an actual parameter in a subprogram or entry call, and the formal parameter is constrained;
 - an actual parameter in a generic instantiation, and the formal parameter is constrained;
 - the return expression in a function whose return type is constrained;
 - the initialization expression of a constant, variable, or formal parameter (of a subprogram, entry, or generic unit) when the type of the constant, variable, or parameter is constrained;
 - the expression of an enclosing record or array aggregate, and the expression gives the value of a record or array component (which is necessarily constrained).

Implementation Guideline: Include a case where the aggregate is the value of more than one component, each of which have different bounds (see IG 4.3/T7).

Implementation Guideline: A multidimensional positional aggregate should be tried in some of the above cases.

- T6. Check that the bounds of a null array aggregate are determined by the bounds specified by the choices. In particular, check that the upper bound is not the predecessor of the lower bound.

Check that neither the upper nor the lower bound need belong to the index subtype.

Check that if one choice of a multidimensional aggregate is non-null but the aggregate is a null array, `CONSTRAINT_ERROR` is raised if one of the bounds of the non-null range does not belong to the index subtype.

Implementation Guideline: IG 4.3.2/T11 makes the equivalent check for non-null arrays.

- T7. For a multidimensional aggregate of the form $(F..G \Rightarrow (H..I \Rightarrow J))$, check that:
- H and I may but need not be evaluated if $F..G$ is a non-null range and F or G do not belong to the index subtype;
 - if $H..I$ is a null range, `CONSTRAINT_ERROR` is raised if $F..G$ is a non-null range and F or G do not belong to the index subtype.
 - if no exception is raised, F , G , H , and I are evaluated once, whether or not $F..G$ is a null range.
 - J is evaluated once for each component (zero times if the array is null).
- T8. For a one-dimensional aggregate of the form $(F..G \Rightarrow (H..I \Rightarrow J))$, (i.e., an array of array type rather than a multidimensional array type), check that:
- if $F..G$ is a null range, H , I , and J are not evaluated;
 - if $F..G$ is a non-null range, H and I are evaluated $G-F+1$ times, and J is evaluated $(I-H+1) * (G-F+1)$ times if $H..I$ is non-null.

Perform similar checks for a multidimensional array type that has an array component type, i.e., check that the expression giving the component value is evaluated the appropriate number of times.

- T9. Check that a string literal can be used in a multidimensional aggregate in place of the final one-dimensional aggregate.

Implementation Guideline: Repeat the relevant tests for IG 4.3.2/T5.

Check that the string literal cannot be enclosed in parentheses in such a case.

Implementation Guideline: Include a check that the multidimensional array declaration does not implicitly declare a one-dimensional array of a character type, so "AB" = "AB" is unambiguous.

- T10. Check that a non-aggregate expression in a named component association is evaluated once for each component specified by the association. (Aggregates are checked in T7 and T8.)

- T11. Check that CONSTRAINT_ERROR is raised if a bound in a non-null range of a non-null aggregate does not belong to the index subtype.

Implementation Guideline: IG 4.3.2/T6 makes the equivalent check for null arrays.

Check whether all choices are evaluated before the check is made.

Implementation Guideline: Use a multidimensional aggregate with at least two subaggregates. In one case, a subaggregate should have static bounds; in another case, all subaggregates should have nonstatic bounds.

- T12. Check that CONSTRAINT_ERROR is raised if all subaggregates for a particular dimension do not have the same bounds.

Implementation Guideline: Check for both null and non-null subaggregates.

Check that bounds for subaggregates are determined independently of each other.

Check whether all choices are evaluated before subaggregates are checked for identical bounds.

- T13. Check that "sliding" occurs for a multidimensional aggregate, but not for the component expression of an identical array of array aggregate.

- T14. For a multidimensional aggregate of the form (F..G => ""), check that CONSTRAINT_ERROR is raised if F..G is non-null and F or G do not belong to the index subtype.

Check that the lower bound for a string literal is determined by the applicable index constraint, when one exists (see IG 4.2/T7).

- T15. Check that CONSTRAINT_ERROR is raised for a positional array aggregate whose upper bound exceeds the upper bound of the index base type.

Implementation Guideline: Use both an integer and an enumeration type for the index subtype. In the integer case, NUMERIC_ERROR can be raised when the upper bound exceeds SYSTEM.MAX_INT, but CONSTRAINT_ERROR is preferred (see AI-00387).

- T21. Check that

- a. completeness of an aggregate is not used in overloading resolution.

Implementation Guideline: The aggregate must be simultaneously a legal record aggregate and an incomplete array aggregate. The compiler must consider it ambiguous.

- b. The length of the aggregate is not used for overloading resolution.

- T22. Check that an array aggregate need not be resolvable to a constrained subtype, as shown by the following example:

```
type A is array (INTEGER range <>) of INTEGER
B : BOOLEAN := (1, 2, 3) = A'(1, 2, 3); -- legal
```

- T23. Check that the restrictions on the contexts in which an array aggregate with an others choice may legally appear are not used in overloading resolution.

- T24. Check that a nonstatic choice of an array aggregate can be a range attribute.

4.4 Expressions

Semantic Ramifications

S1. The syntax for expressions containing logical operators does not permit them to be intermixed without using parentheses. For example, A and B or C is illegal and must be written as (A and B) or C or as A and (B or C). Similarly, unless parentheses are used, sequences of exponentiations, such as A ** B ** C, and sequences of unary adding operators, such as -+A or A + -B, are not permitted. Since "abs" is at a higher level of precedence than unary "-", it is legal to write A - abs B. However, since "****" is at a higher level of precedence than "-", it is illegal to write A ** -B. Finally, since "****" and "abs" are at the same level of precedence, it is illegal to write A ** abs(B).

S2. Names that denote types, subtypes, procedures, packages, tasks, entries, exceptions, operators, labels, blocks, or loops are not permitted as primaries in expressions. Similarly, the BASE and RANGE attributes do not have values and hence are not permitted as primaries.

S3. Overloading resolution in expressions is covered in IG 8.7.

S4. It is important to note that any real (or integer) expression can be calculated with *more* precision than that requested. For instance, all computations could be performed double length, with only single-length loading and storing to memory. In addition, an optimizing compiler is permitted to perform some computations at compile time (possibly using hardware that is different from the run-time hardware; the compile-time hardware might compute results more precisely or use different rounding algorithms, so different results could be obtained depending on whether computations are performed at run time or at compile time). Compile-time (and run-time) computations are only required to yield results that fall within the appropriate model interval; they need not produce exactly the same values that would be produced by the target hardware. These points are discussed further in IG 4.10/S.

S5. Each primary has a value only if its evaluation does not raise an exception.

S6. Operator symbols and character literals appearing in expressions are subject to the usual visibility rules (RM 8.3/18). In particular, if a type is declared in a package, its operators are not visible outside the package unless a use clause is applied:

```
package P is
    type INT is range 1..100;
    -- implicit declaration of "+" etc.
end P;

X : P.INT := 3;
Y : P.INT := X + 3;      -- illegal; "+" not visible
use P;
Z : P.INT := X + 3;      -- now legal
```

In, not in, and then, or else and := are not operators; they are notations associated with basic operations (see RM 3.3.3/4). Basic operations are visible throughout their scope (RM 8.3/18). In particular, since assignment is a basic operation, the initialization of X and Y is legal even without the use clause for P. Moreover,

```
B : BOOLEAN := 3 in P.INT;
```

would be legal before or after the use clause for P.

Changes from July 1982

S7. "abs" and "not" are given the same precedence as "****".

s8. Reading of out parameters (or their subcomponents) is not allowed (except for bounds and discriminants).

Changes from July 1980

s9. The membership operation now requires a type mark instead of a subtype indication.

s10. "abs" is now an operator instead of a predefined function.

Legality Rules

L1. Only the following names are permitted as primaries in expressions (RM 4.4/3):

- attributes other than 'BASE and 'RANGE; and
- names that denote objects or values (in particular, identifiers declared in object declarations, parameter declarations, or number declarations, indexed components, slices, selected components, and character literals).

L2. Names of formal parameters having mode out are not allowed as primaries in expressions (RM 4.4/3).

L3. Names of subcomponents of formal parameters having mode out are not allowed as primaries, except for names that denote discriminants (RM 4.4/3).

L4. An operator must be directly visible before it can be used in an expression as an operator (RM 8.3/18).

Test Objectives and Design Guidelines

T1. Check that

- a. the logical operators (and, or, xor, and then, or else) cannot be intermixed in expressions unless parentheses are used to separate the different operators.

Implementation Guideline: Try at least the following illegal expressions:

```
A and B or C
A and B and C or else D and E
```

- b. a relation can have at most one relational operator, e.g., a sequence of relations such as $A \leq B \leq C$ is not permitted unless parentheses are used (as in $(A \leq B) \leq C$) or the implicit and is made explicit (as in $A \leq B$ and $B \leq C$).

- c. unless parentheses are used, a simple expression can have at most one unary adding operator, which must precede the leftmost term.

Implementation Guideline: Try at least the following illegal expressions:

```
-+A      A * -B
- -A     A ** - B
A + -C   abs - A
```

Note that `- abs A` is legal.

- d. a factor can have at most one operator, e.g., a sequence of exponentiations such as $A ** B ** C$ is not permitted unless parentheses are used (as in $A ** (B ** C)$). Also, $A ** \text{abs}(B)$ and `not not D` are illegal.

- e. a procedure call cannot be a primary.

T2. Check that type, subtype, subprogram, package, task, entry, exception, label, block, and loop names are not permitted as primaries.

Check that the following attributes are not permitted as primaries: BASE and RANGE.

T3. Check that all operations have the correct precedence.

T4. Check that the form

X in type_mark constraint

is illegal.

4.5 Operators and Expression Evaluation

Semantic Ramifications

S1. Membership tests and short-circuit control forms are basic operations, not operators (RM 3.3.3/4). Since they are not operators, these operations are visible throughout their scope (RM 8.3/18). Operators are visible according to the usual visibility rules (RM 8.3/18), and so a use clause may be needed to make a set of operators directly visible (see IG 4.4/S).

S2. An operator can be invoked using named notation, e.g., "+"(LEFT => A, RIGHT => B).

S3. The rules for evaluation of operands in an expression require evaluation in *some* order, which means evaluation in parallel is not allowed. For example, consider:

F1*F2 + F3

If F1, F2, and F3 are all functions with side effects, then the evaluation orders F1, F3, F2 or F2, F3, F1 are not allowed: F1*F2 is one operand of "+"; if this operand is evaluated before the second operand, it must be completely evaluated before the second operand is evaluated.

S4. In principle, all operands must be evaluated and all operations performed, e.g., in principle

I /= 0 and A/I > 100

must raise NUMERIC_ERROR (or CONSTRAINT_ERROR; see AI-00387) when I = 0 since A/0 raises NUMERIC_ERROR (or CONSTRAINT_ERROR). However, RM 11.6/7 allows predefined operations to be omitted if their only effect is to raise an exception. If an implementation takes advantage of this rule, no exception need be raised by the above expression. However, if A is a function, it must be evaluated whether or not the division is performed. (Of course, A's evaluation can be omitted if an implementation can determine that the evaluation has no effect, i.e., does not raise an exception and does not change any global variables.)

S5. RM 4.5/4 says operators having the same precedence are associated left to right with their operands. This association of operators with operands defines a canonical association. RM 11.6/5 then specifies that different associations are allowed if they produce the same result. For integers, "same result" means that the same value is produced or an exception (NUMERIC_ERROR or CONSTRAINT_ERROR; see AI-00387) is raised. For reals, the result must belong to the model interval defined by the canonical association or an exception (NUMERIC_ERROR or CONSTRAINT_ERROR) must be raised. For example, consider:

F1 + F2 + F3

This expression can be evaluated as if it had been written F1 + (F2 + F3), even if F2 + F3 raises NUMERIC_ERROR (or CONSTRAINT_ERROR; see AI-00387) and the evaluation of (F1 + F2) + F3 would not raise any exception. Because of the possibility of reassociating operators, F3 can be evaluated after F2 and before F1; in such a case, the sum of F2 and F3 must be computed. No reassociation of operands with operators allows F3 to be evaluated after F1 and before F2.

S6. Suppose the evaluation order is F2, F3, "+", F1 and the evaluation of F1 will raise an exception only for this order of evaluation. Since the reassociation of operands with operators is only allowed to introduce a predefined exception (raised by a predefined operation), the suggested evaluation order would not be allowed by RM 11.6/5. In effect, if the primaries of an expression contain calls to user-defined functions, an expression must be evaluated in the canonical order defined by RM 4.5/4-5. These issues are discussed further in IG 11.6/S.

S7. Although, in general, the operators mentioned in RM 4.5 are implicitly declared by type declarations, the "*" operator for fixed point multiplication and the "/" operator for division with two fixed point operands are both declared in STANDARD (RM 4.5.5/9).

S8. If, when computing an operand of an expression, an integer arithmetic operator returns a value that does not belong to the base type, NUMERIC_ERROR (or CONSTRAINT_ERROR; see AI-00387) need not be raised if the mathematically correct result is returned. This can happen, for example, if arithmetic is performed using an operation with a wider range, e.g., if INTEGER arithmetic is performed using LONG_INTEGER operations. RM 11.6/6 allows such substitutions. However, NUMERIC_ERROR (or CONSTRAINT_ERROR; see AI-00387) must be raised for the final result if the expression's value is not a value of the type. For example, consider:

X := (A*B) / C;

Suppose A, B, C, and X are INTEGER variables. NUMERIC_ERROR (or CONSTRAINT_ERROR) need not be raised if A*B lies outside the range of the base type as long as the quotient lies within the range. If the quotient does not lie within the range, then NUMERIC_ERROR (or CONSTRAINT_ERROR) should be raised (by the "/" operation).

S9. NUMERIC_ERROR for real types is discussed in IG 4.5.7/S.

Approved Interpretations

S10. When the RM requires that NUMERIC_ERROR be raised (other than by a raise statement), CONSTRAINT_ERROR can (and should) be raised instead (AI-00387).

Changes from July 1982

S11. "abs" and "not" have the same precedence as "**".

S12. Parameter names for predefined operators are LEFT and RIGHT (not L and R).

Changes from July 1980

S13. Evaluation order rules are specified more completely. Operands cannot be evaluated in parallel.

S14. "abs" is now an operator (instead of a predefined function).

Legality Rules

L1. If named notation is used to call a predefined operator, the operand names are LEFT and RIGHT (RM 4.5/6).

L2. An operator must be directly visible before it can be used in an expression as an operator (RM 8.3/18).

Test Objectives and Design Guidelines

T1. Check that each binary operator can be called using named notation with formal parameter names LEFT and RIGHT (see subsequent tests).

Check that each unary operator can be called using named notation with the formal parameter name RIGHT (see subsequent tests).

4.5.1 Logical Operators and Short-Circuit Control Forms

Semantic Ramifications

S1. The logical operators and short-circuit control forms are implicitly declared for derived boolean types and return a result having the derived type. Relational operators are also declared for derived boolean types, but these operators return a predefined BOOLEAN result (RM 3.5.5/15, RM 4.5.1/2, and RM 4.5.2/3).

Approved Interpretations

S2. If both operands of a predefined logical operator do not have the same number of components, execution of a program is not erroneous (CONSTRAINT_ERROR is raised) (AI-00426).

Changes from July 1982

S3. If both operands of a logical operator are null arrays, the upper bound of the result array is the upper bound of the left operand.

Changes from July 1980

S4. The logical operators now return a value having the same type as their operands when the operand has a derived boolean type or an array type with a derived boolean component type.

S5. The short-circuit control forms now return a value having the same type as their operands when the operand has a derived boolean type.

S6. When the operands are arrays, the lower bound of the result is no longer defined by the index subtype of the array type.

Legality Rules

L1. Both operands of the predefined logical operators must have the same base type (RM 4.5.1/1).

L2. Both operands of the short-circuit control forms must have the same base type (RM 4.5.1/4).

L3. "and", "or", and "xor" are predefined only for boolean types (i.e., the predefined type BOOLEAN and types derived (directly or indirectly) from the predefined BOOLEAN type) and for one-dimensional arrays whose component type is a boolean type (RM 4.5.1/2).

L4. The operands of the short-circuit control forms must have a boolean type (RM 4.5.1/4).

Exception Conditions

E1. CONSTRAINT_ERROR is raised by "and", "or", and "xor" if both operands are arrays having different lengths (RM 4.5.1/3 and RM 4.5.2/7), even if the arrays have undefined component values (AI-00426).

4.5.1.a Logical Boolean Operators

Legality Rules

L1. The operands of predefined "and", "or", and "xor" must both have the same base type (RM 4.5.1/1).

L2. "and", "or", and "xor" are predefined for boolean types (i.e., the predefined type BOOLEAN and types derived (directly or indirectly) from predefined BOOLEAN) (RM 4.5.1/2).

Test Objectives and Design Guidelines

- T1. Check the correct operation of **and**, **or**, and **xor**, including combinations with **not**.

Implementation Guideline: Use deMorgan's law, i.e.,

not (A and B) = not A or (not B)
 not (A or B) = not A and (not B)
 not (not A and (not B)) = A or B

as well as simpler tests.

Implementation Guideline: Include some calls using named notation for parameters.

Check that types derived from **BOOLEAN** are permitted.

- T2. Check that the operands of predefined **and**, **or**, and **xor** cannot have scalar nonboolean types.

Implementation Guideline: Use integer types and nondiscrete types.

4.5.1.b Logical Array Operators

Semantic Ramifications

S1. When a logical operator is applied to two null arrays, the bounds of the result are the bounds of the left operand (RM 4.5.1/3), but when two null arrays are catenated, the right operand determines the bounds of the result (RM 4.5.3/4).

S2. The operands of a logical array operator are checked to see if they have the same number of components. This check is performed before the components are evaluated (AI-00426). Consequently, if some components have undefined values, **CONSTRAINT_ERROR** is raised and execution of the program is not erroneous (see RM 3.2.1/18).

Approved Interpretations

S3. If both operands of a predefined logical operator do not have the same number of components, execution of a program is not erroneous (**CONSTRAINT_ERROR** is raised) (AI-00426).

Legality Rules

- L1. The operands of the predefined logical operators must both have the same base type (RM 4.5.1/1).
- L2. "and", "or", and "xor" are predefined for one-dimensional arrays whose component type is a boolean type (RM 4.5.1/2).

Exception Conditions

- E1. **CONSTRAINT_ERROR** is raised by "and", "or", and "xor" if both operands are arrays having different lengths (RM 4.5.1/3 and RM 4.5.2/7), even if one of the operands has a component with an undefined value (AI-00426).

Test Objectives and Design Guidelines

- T11. Check the correct operation of **and**, **or**, and **xor** for array operands (including combinations with **not**).

Implementation Guideline: Check the truth table for these operations.

Implementation Guideline: Include a check that the operators are defined when the component type is a derived boolean type.

Implementation Guideline: Use named notation for some calls.

- T12. Check that the operations yield correct results for operands having different bounds but the same length.

Check that the bounds of the result are the bounds of the left operand, using each of the contexts in which the bounds of the result have a detectable effect.

Implementation Guideline: These contexts are: initialization of an unconstrained constant array; actual parameter when the formal parameter is unconstrained (both for subprograms and generic instantiations); allocating a constrained array (exception should be raised if bounds are not correct); return statement of a function; qualified expression (exception should be raised); value of a component in an aggregate; default initial value of a record component; default initial value of a constrained formal parameter.

Implementation Guideline: Repeat some of the tests when the result is a null array.

- T13. Check that CONSTRAINT_ERROR is raised when the operands have different lengths.
- T14. Check that the operations are defined for arrays having both INTEGER'LAST components and more than INTEGER'LAST components.

Check that the operations are defined for packed arrays.

Implementation Guideline: The type declarations for such arrays might raise NUMERIC_ERROR (or CONSTRAINT_ERROR; see AI-00387). The object declarations might raise STORAGE_ERROR, NUMERIC_ERROR, or CONSTRAINT_ERROR (see IG 3.6/S and AI-00387).

- T16. Check that multidimensional arrays and one-dimensional nonboolean arrays are forbidden as operands.

Implementation Guideline: Check private types whose full declaration declares a derived boolean type.

Check that and then and or else are not defined for one-dimensional BOOLEAN arrays (see IG 4.5.1.c/T21).

4.5.1.c Short-circuit Control Forms

Legality Rules

- L1. The operands of short-circuit control forms must both have the same boolean type (RM 4.5.1/4).

Test Objectives and Design Guidelines

- T21. Check that operands of short-circuit control forms cannot have a nonboolean type.

Implementation Guideline: In particular, check one-dimensional arrays of BOOLEAN and derived boolean types.

- T22. Check that and then and or else are actually short-circuit evaluated.

Implementation Guideline: Use conditions such as

A /= 0 and then B/A > C

or

P /= null and then P.F = D

and check that the appropriate alternative is selected. Also check that no exceptions (such as NUMERIC_ERROR or CONSTRAINT_ERROR) are raised by the evaluation of the condition. Try some conditions with and and or operators as well as short circuited and then or or else.

- T23. Check the truth tables for and then and or else.

Implementation Guideline: Include a check for derived booleans.

4.5.2 Relational Operators and Membership Tests

Semantic Ramifications

S1. Operands of relational operators must have the same base type, but they may have different subtypes. The subtype of one operand does not have to satisfy the subtype constraint of the other operand. For example, if *X* is a variable having subtype *WEEKDAY* (so it can only have the values *MON..FRI*), *X* can be compared with *SUN*, even though *SUN* is not a permitted value of *X*.

S2. The note in RM 4.5.2/12 states that a predefined relational operation never raises an exception although (of course), evaluation of an operand may cause an exception to be raised. This note implies that even if an implementation evaluates *A < B* by computing *A - B* and comparing the result with zero, *NUMERIC_ERROR* (or *CONSTRAINT_ERROR*; see AI-00387) should not be raised (even though RM 11.1/6 suggests that this exception could be raised). A programmer-defined relational operator may, of course, raise an exception.

S3. Although RM 4.5.2/3 says the operand type for equality and inequality may be "any type," other rules in the language imply that equality operations are not *predefined* for all types. In particular, equality operations are not implicitly declared for limited types. Moreover, even if a user-defined equality operator is provided for a limited type, no equality operator is implicitly declared for a composite type having components of the limited type. For example:

```
type T is limited private;
function "=" (X, Y : T) return BOOLEAN;
type R is array (1..10) of T;
```

"=" is not implicitly declared for type *R*. Unless an explicit declaration of "=" is provided, objects having type *R* cannot be compared for equality (or inequality).

S4. For scalar types, the membership tests, *In* and *not In*, are defined in terms of the predefined ordering operators, even if the user has redefined these operators.

S5. A comparison such as *null = null* is legal if there is only one visible equality operator for an access type (since, in such a case, *null*'s type can be determined from the fact that *null* has some access type and from the type required by the visible "=" operator). In particular, the scope of an access type is not directly relevant:

```
package P is
  type STR_ACC is access STRING;      -- declares "=" for STR_ACC
end P;

with P;
package Q is
  type INT_ACC is access INTEGER;      -- declares "=" for INT_ACC
  X : BOOLEAN := null = null;          -- legal; only one "=" visible
  use P;
  Y : BOOLEAN := null = null;          -- illegal; two "="'s visible
end Q;
```

The comments in each case refer to equality operations declared for access types.

Changes from July 1982

S6. There are no significant changes.

Changes from July 1980

S7. Lexicographic order is defined.

Legality Rules

- L1. The operands of the predefined ordering and equality operators must have the same base type (RM 4.5.2/1).
- L2. For a membership operation, the base type of the type mark (and similarly, the base type of each bound in a range, or the base type of a RANGE attribute) must be the same as the base type of the left operand (RM 4.5.2/10).
- L3. Equality (and inequality) are not predefined for limited types (RM 4.5.2/1) or for the type *universal_fixed* (RM 4.5.5/11). (A limited type is a task type, a limited private type, or a composite type containing a subcomponent having a limited type (RM 7.4.4/2)).
- L4. The ordering operators are predefined only for scalar operands and for operands having a one-dimensional array type whose components have a discrete type (RM 4.5.2/1).

Test Objectives and Design Guidelines

(For T1, T2, and T3, see IG 4.5.2.a/T1-T3.)

- T4. Check that relational and membership operations return values of type STANDARD.BOOLEAN, even when this type is hidden.

Implementation Guideline: Try declaring a new type BOOLEAN; check that expressions involving relational and membership operations can be used as conditions, but are not legal expressions of the user-defined BOOLEAN type. Derived boolean types are checked in: IG 4.5.2.c/T20.

- T5. Check that relational expressions of the form $A < B < C$ are forbidden for discrete types.
- T7. Check that equality and inequality are not predefined for limited types or for types derived from a limited type (see also IG 7.4.2/T1).

Implementation Guideline: Check a task type, a limited private type, array of tasks, array of limited private types, record with a task or limited private component, arrays of a limited composite type, and records with a component of a limited composite type.

Implementation Guideline: For composite limited types, try cases where equality is defined for all components of the type.

- T8. Check that relational operators are not defined for operands of different types.
- T9. Check that membership operations are not defined for operands of different types.

Implementation Guideline: Use operands that belong to the same class of types.

Implementation Guideline: Include a check using the RANGE attribute when the index range does not have the correct type.

4.5.2.a Relational and Membership Operations (Enumeration)**Test Objectives and Design Guidelines**

- T1. Check that:

- = and /= produce correct results, in particular, for operands having different subtypes;
- the ordering of enumeration literals, as defined by the ordering operators, is the same as the order of occurrence of the literals in the type definition.

Implementation Guideline: For example, check that

$$(A < B) = (T'POS(A) < T'POS(B))$$

holds for any A and B, where the subtypes of A and B may be different from each other and from T.

Implementation Guideline: Use named notation for some calls of each equality and ordering operator.

- T2. Check the proper operation of the membership operations `in` and `not in`, using subtype names and explicit ranges as the second operand.

Implementation Guideline: Include a use of `A RANGE`.

Implementation Guideline: When a subtype name is used, at least one case should contain a first operand value that lies outside the range associated with the subtype name, e.g., `MON` in `MIDWEEK`;

Implementation Guideline: Try a case where the user has redefined the ordering operators.

Implementation Guideline: Check all forms of membership test:

```
X in type_mark
X in T' RANGE
X in L..R:
```

- T3. Check that variables of two enumeration types having identical sets of enumeration values appearing in the same order cannot be compared with an equality or ordering operator.

Implementation Guideline: For example, given

```
type T is (A, B, C, D);
type U is (A, B, C, D);
X : T;
Y : U;
```

it is illegal to write `X = Y` or `X < Y`.

4.5.2.b Relational and Membership Operations (Character)

Test Objectives and Design Guidelines

- T10. Check that an enumeration type imposing an 'unnatural' order on alphabetic characters, e.g., type `T` is `('C', 'B', 'E', 'D')`; yields the appropriate results for the ordering operators, e.g., `T('C') < T('B')`.

- T11. Check membership tests for an 'unnatural' ordering of character literals.

4.5.2.c Relational and Membership Operations (Boolean)

Semantic Ramifications

S1. When a boolean type is derived, the predefined equality and ordering operations are implicitly declared for the derived type (RM 3.4/6). These operations take operands having the derived boolean type and return a predefined `BOOLEAN` result. The result has a predefined `BOOLEAN` type as specified in RM 4.5.2/3. In particular, the result does not have the derived type:

```
type NB is new BOOLEAN;
A, B : NB := TRUE;
X : BOOLEAN := A = B = B;      -- illegal
Y : BOOLEAN := A = B = TRUE;   -- legal
Z : NB      := A = B;           -- illegal
```

`A = B = B` is illegal because `(A = B)` has type `BOOLEAN`, and there is no equality operator with a `BOOLEAN` left operand and an `NB` right operand. `A = B = TRUE` is legal because `TRUE` can be resolved to the `BOOLEAN` enumeration literal. The initialization of `Z` is illegal because `A = B` does not return an `NB` value.

Test Objectives and Design Guidelines

- T20. Check all combinations of the relational and membership operations for `BOOLEAN` values and ranges.

Implementation Guideline: Include a check for derived boolean types.

- T21. Check that for a derived boolean type, the relational, equality, and membership operations have the result type predefined BOOLEAN.

Implementation Guideline: To ensure that an implementation does not provide, say, an equality that returns a predefined BOOLEAN type and one that returns the derived boolean type, include some cases such as DERIVED_BOOL(A=B=C), which will be unambiguous if the two equality operators are provided.

4.5.2.d Relational and Membership Operations (Integer)

Semantic Ramifications

- S1. Comparisons with integer literals do not raise any exception if the literal belongs to the base type, and need not raise NUMERIC_ERROR (or CONSTRAINT_ERROR; see AI-00387) even if the literal has a value outside the base type:

```
type T is range 0..3000;
X : T := 15;
C : BOOLEAN := X = 1_000_000;    -- NUMERIC_ERROR?
```

If T'BASE'LAST is less than 1_000_000, then the implicit conversion of 1_000_000 to T'BASE can raise NUMERIC_ERROR (or CONSTRAINT_ERROR; see AI-00387) (RM 4.5/7). However, the exception need not be raised, because in the above case, the answer is obviously FALSE if 1_000_000 exceeds T'BASE'LAST; RM 11.6/6 explicitly states that NUMERIC_ERROR need not be raised in such a case.

- S2. Relational operators are defined to take arguments belonging to the base type. For example, consider:

```
type T is range -12..10;
X : T := -3;
B : BOOLEAN := X = 12;    -- no exception raised
```

No exception can be raised by the equality operator because the base type of T at least includes the symmetric range -12..12, so 12 belongs to the base type even though it does not belong to T.

Approved Interpretations

- S3. When the RM requires that NUMERIC_ERROR be raised (other than by a raise statement), CONSTRAINT_ERROR can (and should) be raised instead (AI-00387).

Test Objectives and Design Guidelines

- T31. Check that the relational and membership operations yield correct results for integer types.

Implementation Guideline: Include a use of A'RANGE.

Implementation Guideline: Separately test SHORT_INTEGER, INTEGER, LONG_INTEGER, other predefined integer types (if any), and derived integer types.

Implementation Guideline: For some of the membership tests, include cases when the ordering operators have been redefined.

Implementation Guideline: Use named notation once for each of the equality and ordering operators.

Implementation Guideline: Check all forms of membership test:

```
X in type_mark
X in T'RANGE
X in L..R;
```

- T32. Check whether NUMERIC_ERROR (or CONSTRAINT_ERROR; see AI-00387) is raised when a comparison (or the first operand in a membership test) is a literal exceeding the range of the base type.

Implementation Guideline: If an implementation supports more than one predefined integer type, distinguish cases where the literal exceeds `SYSTEM.MAX_INT` and cases where the literal is less than `SYSTEM.MAX_INT`.

Check that no exception is raised when a literal is used that belongs to the base type but lies outside the subtype of the other operand.

Implementation Guideline: Use values such that overflow could occur if the relational result is computed by subtracting one operand from the other.

4.5.2.e Relational and Membership Operations (Fixed/Float)

Semantic Ramifications

S1. "not" ($A < B$) need not be equal to ($A \geq B$) under certain circumstances, e.g., when the model intervals associated with A and B have more than one value in common (see RM 4.5.7). This is a natural consequence of the rules specified for real computations; because of these rules, the values of A and B are, for nonmodel numbers, implementation-defined within certain limitations. This potential imprecision makes the results of comparisons implementation-dependent.

S2. Ada does impose a language-defined relationship between the results of "=" and "/=", i.e., Ada does require that $\text{not } (A = B) = (A \neq B)$ (RM 4.5.2/2). Similarly, in and not in have a complementary relationship (RM 4.5.2/10).

Test Objectives and Design Guidelines

T41. For floating point types, check the following:

- $A \neq B$ same as $\text{not } (A = B)$, even for numbers that are not model numbers,
- $A < B$ same as $\text{not } (A \geq B)$, for model numbers,
- $A > B$ same as $\text{not } (A \leq B)$, for model numbers,
- adjacent model numbers give correct results,
- nonmodel numbers with distinct model intervals give correct results,
- when the intersection of two model intervals is a single model number, the correct result is given.

Implementation Guideline: Use named notation at least once for each equality and ordering operator.

T42. For floating point types, check whether `NUMERIC_ERROR` (or `CONSTRAINT_ERROR`; see AI-00387) is raised when a literal used in a comparison or as the first operand of a membership operation does not belong to the base type.

Check that no exception is raised for a floating point relational operator or for a membership operation if literal values belong to the base type.

Implementation Guideline: Include values such that overflow would occur if the relational operation were implemented using subtraction.

T43. Check that $A \text{ in } T$ and $A \text{ not in } T$ give the appropriate results even when user-defined ordering operators exist for T .

Implementation Guideline: Check both forms of membership test:

```
X in type_mark
X in L..R;
```

T51. For fixed point types:

- $A \neq B$ same as $\text{not } (A = B)$, even for numbers that are not model numbers,

- b. $A < B$ same as not $(A \geq B)$, for model numbers,
- c. $A > B$ same as not $(A \leq B)$, for model numbers,
- d. adjacent model numbers give correct results,
- e. nonmodel numbers with distinct model intervals give correct results,
- f. when the intersection of two model intervals is a single model number, the correct result is given.

Implementation Guideline: Use named notation at least once for each equality and ordering operator.

- T52. For fixed point types, check whether `NUMERIC_ERROR` (or `CONSTRAINT_ERROR`; see AI-00387) is raised when a literal used in a comparison or as the first operand of a membership operation does not belong to the base type.

Check that no exception is raised for a fixed point relational or a membership operation if literal values belong to the base type.

Implementation Guideline: Include values such that overflow would occur if the relational operation were implemented using subtraction.

- T53. Check that $A \text{ in } T$ and $A \text{ not in } T$ give the appropriate results, even when user-defined ordering operators exist for T .

Implementation Guideline: Check both forms of membership test:

```
X in type_mark
X in L..R;
```

4.5.2.f Relational and Membership Operations (Array)

Semantic Ramifications

- S1. When a subcomponent of an array has a record type with default discriminants, equality comparisons must generally be performed on a component-by-component basis. For example:

```
subtype SMALL is INTEGER range 0..2000;
type VSTRING (CUR_LEN : SMALL := 0) is
  record
    VAL : STRING (1..CUR_LEN);      -- or a variant part
  end record;
type VARRAY is array (1..10) of VSTRING;

X : VARRAY;

X := (1..10 => (5, "AAAXX"));
Y := X;

X(1) := (3, "CCC");
Y(1) := (3, "CCC");

if X = Y then ...
```

The comparison, $X = Y$, must yield the value `TRUE` even though the position occupied by $X(1).VAL(4)$ probably equals 'X' and $Y(1).VAL(4)$ probably equals 'Y' for most implementations. Alternatively, the assignments of $(3, \text{"CCC"})$ must set the unused positions of the `VAL` component to a specific value. If this approach is taken, then component-by-component comparison is not necessary. The situation illustrated by this example occurs whenever an

array component type contains (directly or indirectly) a component whose discriminant values can be changed by assignment.

S2. Two null arrays are equal even if corresponding dimensions do not have the same bounds. In particular, if a dimension of one null array specifies a null range, the corresponding dimension of the other null array need not be null. (Of course, some other dimension of the second array must be null.) If A and B are null arrays, $A < B$ is false since $A = B$.

S3. If the component type of a one-dimensional array is a discrete type with a user-defined ordering operator, the user-defined operator is not used when computing the lexicographic ordering relationship (RM 4.5.2/9). Note that lexicographic order is defined even for arrays having noncharacter component types, e.g., arrays of integers.

S4. For arrays, the membership test `In` checks whether the value of the expression belongs to the subtype denoted by the type mark, i.e., it checks to see if the value satisfies the constraints of the subtype indication (RM 3.3/4):

- if the subtype indication is a type mark for an unconstrained array, the test returns TRUE.
- if the subtype indication is for a constrained array, then the test returns TRUE if and only if corresponding dimensions of the value and the subtype have the same bounds (RM 3.6.1/4).

Since corresponding dimensions of arrays need not have the same bounds in equality comparisons, $A = B$ can be true even though $A \text{ In } T$ is true and $B \text{ In } T$ is false:

```
X : STRING (1..0);
Y : STRING (5..3);
subtype NULL_STRING is STRING (1..0);
... X = Y                -- true
... X in NULL_STRING     -- true
... Y in NULL_STRING     -- false
```

Note that `in` gives the result FALSE if and only if a qualified expression using the type mark would raise CONSTRAINT_ERROR.

S5. If two arrays have different lengths, they can be compared for equality without raising an exception (as long as all components have defined values; see RM 3.2.1/18). In particular, an exception should not be raised when comparing null arrays even though some ways of computing the length might raise an exception. For example: $X(1..\text{INTEGER'FIRST}) = Y$ should not raise an exception even though, when computing the length of the slice, an implementation might attempt to compute $\text{INTEGER'FIRST} - 1$, which would overflow. An implementation should first check to see if the upper bound is less than the lower bound, and only then should it attempt to compute the length.

Test Objectives and Design Guidelines

T61. Check that the ordering relations are not predefined for multidimensional arrays or for one-dimensional arrays of nondiscrete types.

Implementation Guideline: For the one-dimensions' case, include a component type that is a one-dimensional array of a discrete type, as well as a nondiscrete component type for which user-defined ordering operators have been provided.

T62. Check that ordering comparisons yield correct results for one-dimensional discrete array types.

Implementation Guideline: Try the following combinations: null array operands; one operand a null array; operands with an identical number of components and different bounds (include one-component arrays as well as N-component arrays; include cases where all components are equal and cases where all but the last

component is equal); operands with different (nonzero) numbers of components (include cases where the shorter operand is identical to the longer operand).

Implementation Guideline: Try STRING types, arrays of integers, and arrays of an enumeration type other than CHARACTER.

Implementation Guideline: Repeat portions of this test when user-defined ordering operators are available for the discrete component type.

Implementation Guideline: Use named notation at least once for each ordering operator.

- T64. Check that equality comparisons yield correct results for one-dimensional and multidimensional array types.

Implementation Guideline: Use both null and non-null arrays with identical bounds, different bounds and the same length, and bounds with different lengths. Also for null arrays, try cases where corresponding dimensions are not both null.

Implementation Guideline: Check a case where the array has a bound that depends on a discriminant with defaults; ensure that only the correct components are compared (see IG 4.5.2.f/S).

Implementation Guideline: Use named notation at least once for each equality operator.

Check that comparing arrays of different lengths does not raise an exception.

Implementation Guideline: Include a null slice with bounds 1..INTEGER'FIRST.

- T65. Check that the In and not In operations yield correct results for one-dimensional and multidimensional array types when:

- the type mark denotes an unconstrained array type mark;
- the type mark denotes a constrained array.

Implementation Guideline: Include a case where $A = B$ but A and B do not belong to the same subtype.

Implementation Guideline: Use both null and non-null array operands and subtype indications.

Implementation Guideline: Use arrays with components of a limited type as well as nonlimited types.

4.5.2.g Relational and Membership Operations (Record)

Semantic Ramifications

S1. For most record types, equality can be implemented without considering the internal structure of the record object. However, equality comparisons must make sure that unused fields of a record's representation have the same value if whole-object comparison is to work correctly. There is one kind of record type for which component comparison must usually be used because currently unused parts of a record value can contain information from a previous value. For example, consider a record type having components of a varying length string type:

```

subtype TWO_FIVE_FIVE is INTEGER range 0..255;
type VSTRING (CUR_LEN : TWO_FIVE_FIVE := 0) is
  record
    VAL : STRING(1..CUR_LEN);
  end record;

type TWO_STR is
  record
    A, B : VSTRING;
  end record;

X, Y : TWO_STR;
...

```

```

X := ((5, "AAAXX"), (5, "BBYY"));
Y := ((5, "CCCZZ"), (5, "BBYY"));
X.A := (3, "HHH");
Y.A := (3, "HHH");
if X = Y then

```

X = Y is TRUE, but a careless implementation could easily yield FALSE in this case by comparing the no longer valid values of X.A.VAL(4) and X.B.VAL(4), namely, 'X' and 'Z'. (Note that X.A.CUR_LEN is 3, not 4; since the value X.A.VAL(4) no longer exists, it must be ignored when comparing X and Y.) In short, when a component of a record has a type with default discriminants, a whole-object comparison of the complete record value will not necessarily yield correct results.

S2. A similar situation arises with the 'CONSTRAINED' attribute. Two record values can be equal even though their CONSTRAINED attributes have different values. If an implementation has chosen to store the value of this attribute in the record itself, the value must be ignored in equality comparisons.

S3. For a record type T without discriminants, R in T is always TRUE. If T is an unconstrained type with discriminants, then R in T is also always TRUE. If T is a constrained record type, then R in T is TRUE if and only if R's discriminant values equal the discriminant values imposed by T's discriminant constraint.

Legality Rules

L1. The ordering operators are not predefined for record types (RM 4.5.2/3).

Test Objectives and Design Guidelines

T71. Check that equality and inequality are evaluated correctly for records whose components do not have changeable discriminants.

Implementation Guideline: Try some record types that are likely to have unused bits in some portions of their representation.

Implementation Guideline: Use named notation at least once for each equality operator.

T72. Check that equality and inequality are evaluated correctly for records with changeable discriminants, including records designated by access values.

T73. Check that equality and inequality are evaluated correctly for record objects having different values of the 'CONSTRAINED' attribute.

T74. Check that the membership test In (not In) always yields TRUE (FALSE) for record types, private types, and limited private types without discriminants or for unconstrained types with discriminants.

Check that the membership operator In (not In) yields TRUE if the discriminants of the left value equal the discriminants of the subtype indication.

Implementation Guideline: Check record, private, and limited private types.

4.5.2.h Relational and Membership Operations (Access)

Semantic Ramifications

S1. The value null for different access subtypes is the same value, and hence compares as being equal:

```

type PERSON (SEX : GENDER) is record ... end;

```

```

type PERSON_NAME is access PERSON;
subtype MALE is PERSON_NAME(M);
subtype FEMALE is PERSON_NAME(F);

```

```

A      : MALE;                      -- null default value
B      : FEMALE;
C      : PERSON_NAME;
EQUAL  : BOOLEAN := A = B;          -- is TRUE; so is A = C

```

S2. The membership test checks whether a value belongs to a type, i.e., for access types, in returns TRUE if:

- the access value is the null value, or
- the designated subtype is a scalar type, an access type, a task type, a record or private type without discriminants, or an unconstrained array, record, or private type (AI-00324), or
- the designated subtype is a constrained type with discriminants and the designated object has the same discriminants, or
- the designated subtype is a constrained array type and the designated object has the same index bounds.

S3. For access types designating arrays, the situation is a bit more complicated since such types need not be specified with index constraints, e.g.:

```

type STR_NAME is access STRING;

S : STR_NAME (1..15);
... S in STR_NAME ...

```

This membership test will always yield TRUE, since the value of S will always either be null or will designate an object of type STRING.

S4. The other possibility for access types designating arrays is that an index constraint is imposed on the access type. In this case, in yields TRUE if the value of the first operand is null, or if it designates an array object whose bounds equal those specified for the type mark. For example:

```

subtype NAME is STR_NAME (1..15);
...
... S in NAME ...          -- yields TRUE even if S = null

```

S5. For access types designating records (or private types with discriminants), the situation is similar and is unaffected by whether default values are specified for the discriminants. In essence, in yields TRUE if the value of the first operand is null or if it designates an object whose discriminant values equal the discriminant values specified for the type mark. If the type mark denotes an unconstrained record type, then in always yields TRUE.

```

A in PERSON_NAME      -- always TRUE (see IG 4.5.2.b/S1)
A in MALE              -- always TRUE since A is constrained
C in MALE              -- TRUE if C = null or C.SEX = M

```

Approved Interpretations

An access value of type T belongs to every subtype of T if T's designated type is neither an array type nor a type with discriminants (AI-00324).

Test Objectives and Design Guidelines

T81. Check that equality and inequality are evaluated correctly for access values.

Implementation Guideline: Include a case where the access values designate different objects having the same value.

Implementation Guideline: Use named notation at least once for each equality operator.

T82. Check that in and not in operators are evaluated correctly for access types when the designated subtype is:

- a. a scalar type;
- b. an array type (constrained or unconstrained);
- c. a record, private, or limited private type without discriminants;
- d. a record, private, or limited private type with discriminants (with and without default values); where the type mark denotes a constrained and unconstrained type;
- e. a task type.

Implementation Guideline: Check null and non-null values.

4.5.2.i Relational and Membership Operations (Private/Ltd)**Semantic Ramifications**

S1. For a private or a limited type without discriminants, the in membership test always yields the value TRUE.

Test Objectives and Design Guidelines

T91. Check that the membership tests yield correct results for task types, limited private types (see IG 4.5.2.g/T74), composite limited types (see IG 9.5.2.f/T65 and IG 4.5.2.g/T74), and private types without discriminants (see IG 4.5.2.g/T74).

4.5.3 Binary Adding Operators**Semantic Ramifications**

S1. The addition and subtraction operators are defined to take arguments belonging to a base type and to return values of the same base type. For example, consider:

```
type T is range -12..10;
X : T := -3;
Y : T := X + 11;           -- no exception; Y = 8
```

Even though 11 does not belong to the subtype T, T's base type covers at least the symmetric range -12..12, so 11 belongs to T's base type. Since the value of X + 11 belongs to T, no exception will be raised. Similarly:

```
Z : T := Y - X - 2;       -- no exception; Y-X = 11
```

No exception is raised for the result of the first "-" or for the operand of the second "-" even though the value of Y-X does not belong to subtype T.

Changes from July 1982

- S2. The bounds of the result of catenation are determined by the bounds of the right operand if the left operand is a null array (including when the right operand is a null array).
- S3. **CONSTRAINT_ERROR** is raised when a scalar operand for catenation has a value that does not belong to the component subtype of the result.

Changes from July 1980

- S4. The bounds of the result of catenation are defined differently.
- S5. **CONSTRAINT_ERROR** is no longer raised if the result of catenation is a null array.
- S6. The catenation operator can be used when both operands have a scalar type.

Legality Rules

- L1. The operands of the predefined "+" and "-" operations must have the same numeric base type (RM 4.5.3/2).
- L2. The base types of the operands of the predefined "&" operator must not be limited (RM 4.5.3/1) and must either be (RM 4.5.3/2):
- the same type, or
 - one operand must have a one-dimensional array type whose component type is C and the base type of the other operand must be C.

Exception Conditions

See individual subsections.

Test Objectives and Design Guidelines

- T1. For integer, fixed point, and floating point types, check that "+" and "-" are not predefined for operands having different base types.

Check that "+" and "-" are not predefined for non-numeric types.

Implementation Guideline: Check using operands of a parent and a derived numeric type, an integer plus a noninteger discrete type, different predefined integer types, etc.

- T2. Check that "&" is not defined for multidimensional arrays having the same type or for one-dimensional arrays related by derivation (including derived index types).

4.5.3.a Integer Adding Operators**Semantic Ramifications**

S1. **NUMERIC_ERROR** (or **CONSTRAINT_ERROR**; see AI-00387) need not be raised for a subexpression when the result of an addition or subtraction lies outside the base type, if the result of the complete expression is mathematically correct (RM 11.6/6). For example, when computing $A + B - C$, the sum might overflow, but the overall result might lie within the base type. If an implementation computes the sum in an overlength register, no overflow will actually occur and the correct result can be obtained.

S2. Note that **INTEGER'SUCC** (**INTEGER'LAST**) raises **CONSTRAINT_ERROR** (see RM 3.5.5/8), but **INTEGER'LAST + 1** raises either **NUMERIC_ERROR** or **CONSTRAINT_ERROR** (AI-00387).

Approved Interpretations

S3. When the RM requires that `NUMERIC_ERROR` be raised (other than by a raise statement), `CONSTRAINT_ERROR` can (and should) be raised instead (AI-00387).

Legality Rules

- L1. The operands of the predefined integer "+" and "-" operations must have the same integer base type (RM 4.5.3/2).

Exception Conditions

- E1. `NUMERIC_ERROR` (or `CONSTRAINT_ERROR`; see AI-00387) is raised if the mathematically defined result lies outside the range of the base type (RM 4.5/7).

Test Objectives and Design Guidelines

- T3. Check that "+" and "-" yield the correct results for all predefined integer types.
Implementation Guideline: Check single values as well as values yielding 'LAST and 'FIRST of the base type.
Implementation Guideline: Attempt to declare a type that might be represented as an unsigned integer. Check that neither `NUMERIC_ERROR` nor `CONSTRAINT_ERROR` (see AI-00387) are raised if intermediate negative values are produced.
Implementation Guideline: Use named notation for some calls.
- T4. Check that `NUMERIC_ERROR` (or `CONSTRAINT_ERROR`; see AI-00387) is raised by "+" and "-" for all predefined integer types when the result is outside the range of the base type.
Implementation Guideline: Include cases such as `INTEGER'LAST + N` and `FLOAT(INTEGER'LAST+1)`.

4.5.3.b Floating Point Adding Operators**Semantic Ramifications**

S1. The rules of real addition are a consequence of the semantics for all operations giving a real result (see RM 4.5.7). Hence the testing required is determined by the relationship between the two operands, the result, and the corresponding model intervals. The rules allow computations to be performed with an "overlength accumulator," i.e., with more precision in the registers than the stored values (or the model numbers) demand.

S2. A computer may have a single instruction to extend the precision of two operands and perform the double-length operation. For example, consider:

```

declare
    D1, D2, D3 : DOUBLE;
    A, B, C    : SINGLE;
begin
    -- calculate A and B
    C := A + B;
    D1 := DOUBLE(A + B);
    D2 := DOUBLE(A) + DOUBLE(B);
    D3 := DOUBLE(C);
end;
```

D1 does not necessarily equal D2. A simple compiler will use single length addition for D1 and then convert to double length. However, it could use double-length addition (which could be advantageous if this can be done in a single instruction) and hence get the same value as D2. The code generated need not be consistent. For instance, D1 need not equal D2 or D3.

S3. An optimizer must not treat floating point addition as associative. To check for such an error, write:

```
D := B + C;
E := A + B + C;
```

Let $A = -B$, let A , B , and C be model numbers, and let the model interval for $B + C$ include B (e.g., give C a value that requires one more bit in B 's mantissa if the sum $B + C$ is to be computed exactly). Then $A + B$ must equal zero (since A and $-B$ are model numbers), and E must equal C . However, $D = B$ might evaluate to FALSE; if an optimizer computed $A + B + C$ as $A + D$, it could generate an incorrect result such that E is not equal to C .

S4. For a discussion of when the exception `NUMERIC_ERROR` can be raised for real operations, see IG 4.5.7/S.

Approved Interpretations

S5. When the RM requires that `NUMERIC_ERROR` be raised (other than by a raise statement), `CONSTRAINT_ERROR` can (and should) be raised instead (AI-00387).

Legality Rules

L1. Operands of the predefined floating point "+" and "-" operations must have the same floating point base type (RM 4.5.3/2).

Exception Conditions

- E1. For floating point types, `NUMERIC_ERROR` (or `CONSTRAINT_ERROR`; see AI-00387) is raised if the result of predefined addition or subtraction lies outside the range of the operand's base type and `MACHINE_OVERFLOW`s is true for the base type (RM 4.5.7/7).
- E2. For floating point types, `NUMERIC_ERROR` (or `CONSTRAINT_ERROR`; see AI-00387) can be raised if the result of predefined addition or subtraction lies outside the range of safe numbers for the operand type (RM 4.5.7/7).

Test Objectives and Design Guidelines

T21. For floating point types, check that the operators "+" and "-" produce correct results when:

- a. A , B , $A+B$, and $A-B$ are all model numbers.
- b. A is a model number but B , $A+B$, and $A-B$ are not.
- c. A , B , $A+B$, and $A-B$ are all model numbers with different subtypes.
- d. A and B are model numbers with different subtypes, but $A+B$ and $A-B$ are not model numbers.

Implementation Guideline: Use named notation for some calls to "+" and "-".

T22. Check that `NUMERIC_ERROR` (or `CONSTRAINT_ERROR`; see AI-00387) is raised if `MACHINE_OVERFLOW`s is true and the result of an addition or subtraction lies outside the range of the base type.

Implementation Guideline: If `MACHINE_OVERFLOW`s is not true, report whether `NUMERIC_ERROR` or `CONSTRAINT_ERROR` is raised when the expression occurs as an operand of a relational operator.

Check whether `NUMERIC_ERROR` (or `CONSTRAINT_ERROR`; see AI-00387) is raised if `MACHINE_OVERFLOW`s is true and the result is outside the range of safe numbers, but within the range of the base type (see IG 4.5.7/T1).

T23. Check that nonassociativity of real arithmetic is preserved, even when optimization would benefit if floating point addition were associative.

4.5.3.c Fixed Point Adding Operators

Semantic Ramifications

Approved Interpretations

S1. When the RM requires that `NUMERIC_ERROR` be raised (other than by a raise statement), `CONSTRAINT_ERROR` can (and should) be raised instead (AI-00387).

Legality Rules

L1. The operands of the predefined fixed point "+" and "-" operations must have the same fixed point base type (RM 4.5.3/2); the type cannot be *universal_fixed* (RM 4.5.5/11).

Exception Conditions

E1. For fixed point types, `NUMERIC_ERROR` (or `CONSTRAINT_ERROR`; see AI-00387) is raised if the result of a predefined addition or subtraction lies outside the range of the operand's base type and `MACHINE_OVERFLOW`s is true for the base type (RM 4.5.7/7).

E2. For fixed point types, `NUMERIC_ERROR` (or `CONSTRAINT_ERROR`; see AI-00387) can be raised if the result of a predefined addition or subtraction lies outside the range of safe numbers for the operand type (RM 4.5.7/7).

Test Objectives and Design Guidelines

T31. For fixed point types, check that the operators "+" and "-" produce correct results when:

- A, B, A+B, and A-B are all model numbers.
- A is a model number but B, A+B, and A-B are not.
- A, B, A+B, and A-B are all model numbers with different subtypes.
- A and B are model numbers with different subtypes, but A+B and A-B are not model numbers.

Implementation Guideline: Use named notation for some calls to "+" and "-".

T32. Check that `NUMERIC_ERROR` (or `CONSTRAINT_ERROR`; see AI-00387) is raised for "+" and "-" if `MACHINE_OVERFLOW`s is true and the result lies outside the range of the base type.

Implementation Guideline: Report whether `NUMERIC_ERROR` is raised even if `MACHINE_OVERFLOW`s is not true.

4.5.3.d Array Adding Operators (Catenation)

Semantic Ramifications

S1. Catenation is defined for any nonlimited component type, including array, record, and private types. For example:

```
type INT_ARR_TYPE is array (POSITIVE range <>) of INTEGER;
ARR : INT_ARR_TYPE(1..2) := 1 & 2;      -- same as (1, 2);
```

S2. When catenating two null arrays, the lower bound of the result is given by the right operand (RM 4.5.3/4). When applying a logical operator to two null arrays, the lower bound of the result is given by the left operand (RM 4.5.1/3).

S3. `CONSTRAINT_ERROR` is raised if an operand value does not belong to the result's component subtype (RM 4.5.3/6);

```

subtype UPPER_CASE_LETTERS is CHARACTER range 'A'..'Z';
type UPPER_CASE_STRING is
  array (POSITIVE range <>) of UPPER_CASE_LETTERS;
UPPER : UPPER_CASE_STRING (1..2) := 'B' & '3';      -- CONSTRAINT_ERROR

```

CONSTRAINT_ERROR is raised since '3' does not belong to UPPER_CASE_LETTERS.

S4. The catenation operators can be overloaded to take operands having the same component type, e.g., in the previous example, "&" was implicitly declared to take operands of type CHARACTER, producing a result of type UPPER_CASE_STRING. The catenation operator is also defined for operands of type CHARACTER, producing a result of type STRING. The normal overloading resolution rules determine which operator is to be used:

```

STRING_VAR : STRING (1..2)           := 'B' & 'C';  -- STANDARD."&"
UPPER_VAR  : UPPER_CASE_STRING (1..2) := 'B' & 'C';

```

S5. The length of the result of catenation need not be known at compile time. It is usually undesirable to allocate the maximum possible length for a catenation result. One fairly well-known implementation technique for minimizing storage is to put the result on the end of the stack, adding space as the result is computed.

Legality Rules

L1. The base types of the operands of the predefined "&" operator must not be limited (RM 4.5.3/1) and must either be (RM 4.5.3/2):

- the same type, or
- one operand must have a one-dimensional array type whose component type is C and the base type of the other operand must be C.

Exception Conditions

- E1. CONSTRAINT_ERROR is raised if the upper bound of the catenation result exceeds the upper bound of the index subtype and the result is not a null array (RM 4.5.3/6).
- E2. CONSTRAINT_ERROR is raised if an operand of "&" has the type of the array component but does not belong to the array component's subtype (RM 4.5.3/6).

Test Objectives and Design Guidelines

T41. Check that catenation is not defined for multidimensional arrays or for arrays having a limited component type.

T42. Check that the result of catenating two non-null operands has the lower bound of the left operand when the left operand has the same type as the result, and the lower bound of the index subtype when the left operand has the component type of the result.

Implementation Guideline: There are four cases: array & array, component & array, array & component, component & component. Also, the component type can itself be an array type.

Implementation Guideline: When the left operand is an array, its lower bound should not equal either the lower bound of the index subtype or the lower bound of the right operand.

Implementation Guideline: Use the result as an actual parameter when the formal parameter is an unconstrained array type or as the result of a function call that has an unconstrained return type.

Implementation Guideline: Use named notation for some calls of each of the four catenation operators.

T43. Check that when the left operand is a null array, and

- the right operand is an array (null or non-null) having the type of the result, the bounds of the result are the bounds of the right operand.

- the right operand is a component value, the lower bound of the result is the lower bound of the index subtype.

Check that when the right operand is a null array, and

- the left operand is a non-null array having the type of the result, the bounds of the result are the bounds of the left operand.
- the left operand is a component value, the lower bound of the result is the lower bound of the index subtype.

T44. Check that the correct result is produced when a function returns the result of a catenation whose bounds are not defined statically.

Implementation Guideline: Consider catenating the result of recursive function calls in the return statement of a function, when the length of each catenation is not statically defined.

T45. Check that CONSTRAINT_ERROR is raised if the upper bound of a non-null catenation result would lie outside the range of the index subtype.

Implementation Guideline: The length of the result should not exceed the length allowed by the index subtype.

Check that CONSTRAINT_ERROR is not raised if the length of the result equals the maximum length permitted by the index subtype.

T46. Check that NUMERIC_ERROR is not raised if the length of the result exceeds INTEGER'LAST or SYSTEM.MAX_INT. (Note: CONSTRAINT_ERROR or STORAGE_ERROR should be raised instead.)

T47. Check that catenation is defined for the following component types: records, arrays, private types, and access types.

Check that catenation is defined when the component type is an array type and both operands are arrays.

4.5.4 Unary Adding Operators

Semantic Ramifications

S1. The unary operators are defined for a base type, not a subtype:

```
type T is range 1..10;
X : T := 3;
Y : T := -X + 5;          -- no exception raised
```

No exception can be raised even though -3 does not belong to T, since the result of -X does belong to T's base type (which at least includes the symmetric range -10..10); moreover, -X + 5 does belong to T. In addition, the operators accept any operand value belonging to the base type, e.g.,

```
Z : T := -(1 - X);       -- no exception raised
```

No exception can be raised even though 1 - X = -2, and -2 does not belong to T.

S2. It is possible for the operand of unary "+" to be outside the range of the base type if the operand is an expression, and the expression is evaluated using higher precision operations than required, e.g.,

```
Z := + (X*Y);
```

If the result of X*Y is held in a double length register, as is often the case, then the value might

be greater than that allowed by Z's base type. **NUMERIC_ERROR** (or **CONSTRAINT_ERROR**; see AI-00387) must be raised by "+" if it is not raised by "**".

Approved Interpretations

S3. When the RM requires that **NUMERIC_ERROR** be raised (other than by a raise statement), **CONSTRAINT_ERROR** can (and should) be raised instead (AI-00387).

Changes from July 1982

S4. "abs" and "not" are no longer unary adding operators. (They are highest precedence operators.)

Changes from July 1980

S5. There are no significant changes.

Legality Rules

L1. The operand of predefined unary "+" and "-" must have a numeric type (RM 4.5.4/1).

Test Objectives and Design Guidelines

T1. Check that unary "+" and "-" are not predefined for non-numeric types.

4.5.4.a Integer Unary Adding Operators

Semantic Ramifications

S1. For an integer type, T, **NUMERIC_ERROR** (or **CONSTRAINT_ERROR**; see AI-00387) is only raised for -T'BASE'FIRST if $\text{abs } T'BASE'FIRST > T'BASE'LAST$; -T'BASE'LAST never raises an exception since T'BASE'LAST is in practice never greater than $\text{abs } T'BASE'FIRST$.

Approved Interpretations

S2. When the RM requires that **NUMERIC_ERROR** be raised (other than by a raise statement), **CONSTRAINT_ERROR** can (and should) be raised instead (AI-00387).

Legality Rules

L1. The operand of predefined integer "+" and "-" must have an integer type (RM 4.5.4/1).

Exception Conditions

E1. **NUMERIC_ERROR** (or **CONSTRAINT_ERROR**; see AI-00387) is raised if the result of unary integer "+" or "-" lies outside the range of the base type (RM 4.5/7).

Test Objectives and Design Guidelines

T11. Check that unary "-" and "+" give the correct results for integer operands.

Implementation Guideline: Try all the predefined integer types and derived integer types.

Implementation Guideline: Use named notation for some calls.

T12. Check that **NUMERIC_ERROR** (or **CONSTRAINT_ERROR**; see AI-00387) is raised by unary "-" if the ranges of the base types indicate a nonsymmetric range of values.

T13. Check that for a user-defined integer type, unary "+" and "-" yield and accept results belonging to the base type (rather than the subtype defined by the user).

4.5.4.b Real Unary Adding Operators

Semantic Ramifications

- S1. If A is a model number, then so is $-A$; hence, the operation is exact.
- S2. For a two's-complement representation of floating point or fixed point, `-T'BASE'FIRST` could raise `NUMERIC_ERROR` (or `CONSTRAINT_ERROR`; see AI-00387), but unary minus cannot raise an exception when applied to a model number, since the range of model numbers is symmetric.

Approved Interpretations

- S3. When the RM requires that `NUMERIC_ERROR` be raised (other than by a raise statement), `CONSTRAINT_ERROR` can (and should) be raised instead (AI-00387).

Legality Rules

- L1. The operand of predefined floating point `+` and `-` must have a floating point type (RM 4.5.4/1).
- L2. The operand of predefined fixed point `+` and `-` must have a fixed point type (RM 4.5.4/1).

Exception Conditions

- E1. For floating and fixed point types, `NUMERIC_ERROR` (or `CONSTRAINT_ERROR`; see AI-00387) is raised if the result of a predefined unary `+` or `-` lies outside the range of the operand's base type and `MACHINE_OVERFLOW`s is true for the base type (RM 4.5.7/7).
- E2. For floating and fixed point types, `NUMERIC_ERROR` (or `CONSTRAINT_ERROR`; see AI-00387) can be raised if the result of a predefined unary `+` or `-` lies outside the range of safe numbers for the operand type (RM 4.5.7/7).

Test Objectives and Design Guidelines

- T21. For floating point types, check that:

- $+A$ is equal to A and
- $-(-A)=A$ for model numbers.

Implementation Guideline: Check for digits 5 through 29.

Implementation Guideline: Use named notation for some calls.

- T22. For floating point types, check that `NUMERIC_ERROR` (or `CONSTRAINT_ERROR`; see AI-00387) is raised if `MACHINE_OVERFLOW`s is true for the type and $-A$ exceeds the range of the base type.
- T23. For floating point types, check whether `NUMERIC_ERROR` (or `CONSTRAINT_ERROR`; see AI-00387) is raised for unary `+` or `-` when the operand value lies outside the range of safe numbers but within the range of the base type.

- T31. For fixed point types, check that:

- $+A$ is equal to A and
- $-(-A)=A$ for model numbers.

Implementation Guideline: Use named notation for some calls.

- T32. For fixed point types, check that `NUMERIC_ERROR` (or `CONSTRAINT_ERROR`; see AI-00387) is raised if $-A$ exceeds the range of the base type, and `MACHINE_OVERFLOW`s is true.

- T33. For fixed point types, check whether `NUMERIC_ERROR` (or `CONSTRAINT_ERROR`; see AI-00387) is raised for unary "+" or "-" when the operand value lies outside the range of safe numbers but within the range of the base type.

4.5.5 Multiplying Operators

Semantic Ramifications

Approved Interpretations

- S1. When the RM requires that `NUMERIC_ERROR` be raised (other than by a raise statement), `CONSTRAINT_ERROR` can (and should) be raised instead (AI-00387).

Changes from July 1982

- S2. There are no significant changes.

Changes from July 1980

- S3. Fixed point multiplication and division operators (when both operands have a fixed point type) are declared in `STANDARD`.

- S4. Multiplication or division of a fixed point value by a value of any integer type is no longer predefined. Such operations are only allowed for a fixed point type and the predefined `INTEGER` type.

Legality Rules

- L1. For predefined integer multiplication and division, both operands must have the same integer base type (RM 4.5.5/1).
- L2. For predefined "mod" and "rem", both operands must have the same integer base type (RM 4.5.5/1).
- L3. For predefined floating point multiplication and division, both operands must have the same floating point base type (RM 4.5.4/1).
- L4. For predefined fixed point multiplication, either:
- one operand must have the predefined type `INTEGER` and the other some fixed point type (RM 4.5.5/7); or
 - both operands must have a fixed point type (not necessarily the same fixed point type) (RM 4.5.5/11).

Neither operand can have the type *universal_fixed* (RM 4.5.5/11).

- L5. For predefined fixed point division, either:
- the second operand must have the predefined type `INTEGER` and the first operand must have some fixed point type (RM 4.5.5/7); or
 - both operands must have a fixed point type (not necessarily the same fixed point type) (RM 4.5.5/10).

Neither operand can have the type *universal_fixed* (RM 4.5.5/11).

- L6. The product or quotient of two fixed point types must be explicitly converted to some numeric type (RM 4.5.5/11).
- L7. An operand of a fixed point multiplication or division operation must not be a real literal (see IG 4.5.5.b/S4).

Exception Conditions

- E1. **NUMERIC_ERROR** (or **CONSTRAINT_ERROR**; see AI-00387) can be raised by predefined "*" and "/" if the mathematical result lies outside the range of the base type or outside the range of safe numbers (RM 4.5/7 and RM 4.5.7/7).
- E2. **NUMERIC_ERROR** (or **CONSTRAINT_ERROR**; see AI-00387) is raised by predefined "/", mod, and rem if the second operand has the value zero (RM 4.5.5/12).

4.5.5.a Integer Multiplying Operators**Semantic Ramifications**

- S1. Integer division can never produce a result outside the range of the base type, so division raises **NUMERIC_ERROR** (or **CONSTRAINT_ERROR**; see AI-00387) only for division by zero.
- S2. RM 11.6/6 allows an implementation to compute intermediate results using operations for a type with a wider range than the operands' base type. In particular, when computing an expression such as $A*B/C$, the product, $A*B$, can be held in a double-length register and used directly as the operand for the division operation. In such a case, even though $A*B$ might have a value that lies outside the base type, the result of the division can be computed correctly and can lie within the operands' base type; no exception need be raised.
- S3. Consider a type declaration such as:

type T is range -25..10;

The range of the base type T'BASE must at least be -25..25 (RM 3.5.4/7) and must be the range of some predefined integer type. An expression such as $T'(5)*5/5$ must yield 5 without raising an exception, even though 25 lies outside T's range.

- S4. **INTEGER'FIRST** 1 should not raise any exception, although this computation may be difficult to perform on some machines.

Approved Interpretations

- S5. When the RM requires that **NUMERIC_ERROR** be raised (other than by a raise statement), **CONSTRAINT_ERROR** can (and should) be raised instead (AI-00387).

Legality Rules

- L1. For predefined integer multiplication and division, both operands must have the same integer base type (RM 4.5.5/1).
- L2. For predefined mod and rem, both operands must have the same integer base type (RM 4.5.5/1).

Exception Conditions

- E1. **NUMERIC_ERROR** (or **CONSTRAINT_ERROR**; see AI-00387) is raised for the predefined integer "*" operator if the mathematically defined product of the operand values lies outside the range of the base type (RM 4.5/7).
- E2. **NUMERIC_ERROR** (or **CONSTRAINT_ERROR**; see AI-00387) is raised by predefined "/", mod, and rem if the second operand has the value zero (RM 4.5.5/12).

Test Objectives and Design Guidelines

- T1. Check that the multiplying operators are not predefined for operands having different integer types.

Implementation Guideline: Check operands related by derivation and operands of different predefined integer types.

Check that the multiplying operators are not predefined for operands of an integer and a floating point type, integers and arrays of integers, arrays of integers (matrix multiplication), etc.

Check that `mod` and `rem` are not predefined for real operands of the same type or when the first operand is real and the second is an integer.

- T2. Check that multiplication and division yield correct results.

Implementation Guideline: Use operands with different and identical signs. For division, also check division with and without remainders.

Implementation Guideline: Use named notation for some calls.

- T3. Check that `rem` and `mod` yield correct results.

Implementation Guideline: Use operands with different and identical signs. Check all values for a small value of the second operand.

Implementation Guideline: Use named notation for some calls.

- T4. Check that `NUMERIC_ERROR` (or `CONSTRAINT_ERROR`; see AI-00387) is raised when a product lies outside the range of the base type.

Check that `NUMERIC_ERROR` (or `CONSTRAINT_ERROR`; see AI-00387) is raised when the second operand of `/`, `mod`, or `rem` equals zero.

- T5. Check that multiplication for integer subtypes yields a result belonging to the base type.

4.5.5.b Real Multiplying Operators

Semantic Ramifications

S1. There are effectively four multiplication operations to check (float-float, fixed-fixed, fixed-INTEGER, and INTEGER-fixed) and three for division (float-float, fixed-fixed, and fixed-INTEGER). The difficult cases for both are those for two fixed point operands. The reason for the difficulty is that the potential number of tests is cubed due to the need to consider the type of each operand and the result type separately. In addition, if 'SMALL for both operands is not a power of the same integer, an implementation must take extra care to produce correct results (see IG 3.5.10/S).

S2. If a machine performs division by computing the reciprocal and then multiplying, the result will not generally be within the model interval Ada requires (see IG 4.5.7/S). This means a conforming compiler will not be able use the computer's division instruction.

S3. In principle, since fixed point multiplication or division yields a result having type *universal_fixed*, `NUMERIC_ERROR` can never be raised, except when the divisor is zero (i.e., when the model interval of the divisor includes zero). However, since *universal_integer* results must be converted to some other numeric type, the conversion operation can raise `NUMERIC_ERROR` (or `CONSTRAINT_ERROR`; see AI-00387); it is not possible to tell which operation caused the exception to be raised.

S4. Fixed point multiplication and division by literals is forbidden. Consider for example, `FP (V1 * 3.14)`, where `V1` is a variable having a fixed point type and `FP` is a fixed point type. For the multiplication operation to be defined, both operands must have a fixed point type. The literal 3.14 has the type *universal_real* and can be implicitly converted to any required fixed point type (RM 4.6/15), but the context is insufficient to determine a unique fixed point type since there are always at least two fixed point types whose scope includes any compilation unit. These types are the type `DURATION` (RM 9.6/4 and RM C/19) and the anonymous predefined

fixed point type required by RM 3.5.9/7. Both types are declared in STANDARD, so their scope includes every compilation unit. Since there is no unique fixed point type to which the literal can be converted, the expression is illegal. In general, real literals cannot be used with fixed point multiplication or division operators unless the literals are explicitly qualified by a type mark.

S5. When a fixed point value is divided by an integer, `NUMERIC_ERROR` (or `CONSTRAINT_ERROR`; see AI-00387) is raised if the divisor is zero because the model interval of the result is undefined, not because the divisor is zero. This distinction is only important when suppressing the exception check. `DIVIDE_CHECK` is used to suppress the check for division by zero for integer divide, `rem`, and `mod`. `DIVIDE_CHECK` does not apply to any fixed or floating point operator, including division of a fixed point value by an integer, since the RM does not mention making a special check for division by zero for fixed point division (RM 4.5.5/12).

Approved Interpretations

S6. When the RM requires that `NUMERIC_ERROR` be raised (other than by a raise statement), `CONSTRAINT_ERROR` can (and should) be raised instead (AI-00387).

Legality Rules

- L1. For predefined floating point multiplication and division, both operands must have the same floating point base type (RM 4.5.4/1).
- L2. For predefined fixed point multiplication, either:
 - one operand must have predefined type `INTEGER` and the other some fixed point type (RM 4.5.5/7); or
 - both operands must have a fixed point type (not necessarily the same fixed point type) (RM 4.5.5/11).

Neither operand can have the type *universal_fixed* (RM 4.5.5/11).

- L3. For predefined fixed point division, either:
 - the second operand must have the predefined type `INTEGER` and the first operand must have some fixed point type (RM 4.5.5/7); or
 - both operands must have a fixed point type (not necessarily the same fixed point type) (RM 4.5.5/10).

Neither operand can have the type *universal_fixed* (RM 4.5.5/11).

- L4. The product or quotient of two fixed point types must be explicitly converted to some numeric type (RM 4.5.5/11).
- L5. An operand of a fixed point multiplication or division operation must not be a real literal (see IG 4.5.5 b 34).

Exception Conditions

- E1. `NUMERIC_ERROR` (or `CONSTRAINT_ERROR`; see AI-00387) is raised for predefined floating point multiplication and division if the result lies outside the range of the base type and `MACHINE_OVERFLOW` is true for the base type.
- E2. `NUMERIC_ERROR` (or `CONSTRAINT_ERROR`; see AI-00387) may be raised for predefined floating point multiplication and division if the result is within the range of the base type but outside the range of safe numbers (see RM 4.5.7/7).
- E3. `NUMERIC_ERROR` (or `CONSTRAINT_ERROR`; see AI-00387) is raised for predefined floating point division if the divisor is zero (RM 4.5.7/7).

- E4. **NUMERIC_ERROR** (or **CONSTRAINT_ERROR**; see AI-00387) is raised for fixed point multiplication by an **INTEGER** if the result lies outside the range of the base type (RM 4.5.7/7).
- E5. **NUMERIC_ERROR** (or **CONSTRAINT_ERROR**; see AI-00387) is raised for fixed point division by an **INTEGER** if the divisor is zero (RM 4.5.7/7).

Test Objectives and Design Guidelines

T21. Check that for the predefined floating point **"*"** operator, correct results are produced when:

- A, B, and **A*B** are all model numbers.
- A and B are model numbers, but **A*B** is not.
- A is a model number, but B and **A*B** are not.
- A, B, and **A*B** are all model numbers with different subtypes.
- A and B are model numbers with different subtypes, but **A*B** is not a model number.

Check that for the predefined floating point **"/"** operator, correct results are produced when:

- A, B, and **A/B** are all model numbers.
- A and B are model numbers, but **A/B** is not.
- A is a model number, but B and **A/B** are not.
- A, B, and **A/B** are all model numbers with different subtypes.
- A and B are model numbers with different subtypes, but **A/B** is not a model number.

Implementation Guideline: Use named notation for some calls.

T22. Check that the same floating point type must be used with multiplying operators.

Check that **mod** and **rem** are not predefined for floating point types (see IG 4.5.5.a/T1).

T23. Check that **NUMERIC_ERROR** (or **CONSTRAINT_ERROR**; see AI-00387) is raised if **MACHINE_OVERFLOW** is true and the result of multiplication or division lies outside the range of the base type.

Check that **NUMERIC_ERROR** (or **CONSTRAINT_ERROR**; see AI-00387) is raised when a floating point value is divided by zero, if **MACHINE_OVERFLOW** is true.

Implementation Guideline: If **MACHINE_OVERFLOW** is not true, report whether **NUMERIC_ERROR** (or **CONSTRAINT_ERROR**) is raised in the above cases.

Check whether **NUMERIC_ERROR** (or **CONSTRAINT_ERROR**; see AI-00387) is raised if **MACHINE_OVERFLOW** is true and the result is outside the range of safe numbers, but within the range of the base type (see IG 4.5.7/T1).

T24. Check that no exception is raised when underflow occurs.

Implementation Guideline: In particular, **NUMERIC_ERROR** should not be raised.

Implementation Guideline: Report whether underflow is gradual or not.

T31. Check the following for a variety of fixed point types:

- fixed*integer** when all values are model numbers.
- fixed*integer** values are bounded correctly for nonmodel numbers (check same values for **integer*fixed**).

- c. fixed/integer when all values are model numbers.
- d. fixed/integer values are bounded correctly for nonmodel numbers.

Implementation Guideline: Use named notation for some calls.

T32. Special tests are needed for fixed*fixed and fixed/fixed because of the three types involved. Check for a variety of the three types involved:

- a. fixed*fixed with all values being model numbers.
- b. fixed*fixed with operands as model numbers but results are not.
- c. fixed*fixed with no model numbers.
- d. fixed/fixed with all values being model numbers.
- e. fixed/fixed with operands as model numbers but results are not.
- f. fixed/fixed with no model numbers.

T33. Check that mod and rem are not predefined for fixed point types (see IG 4.5.5.1/T1).

T34. Check that NUMERIC_ERROR (or CONSTRAINT_ERROR; see AI-00387) is raised for multiplication or division of two fixed point values when the result is converted to some numeric type and the value does not belong to the target base type.

Check that NUMERIC_ERROR (or CONSTRAINT_ERROR; see AI-00387) is raised for multiplication of a fixed point value by an integer, or division of a fixed point value by an integer, if the result lies outside the range of the fixed point base type.

Check that NUMERIC_ERROR (or CONSTRAINT_ERROR; see AI-00387) is raised when a fixed point value is divided by zero (either an INTEGER zero or a fixed point zero).

Implementation Guideline: If MACHINE_OVERFLOW is not true, report whether NUMERIC_ERROR (or CONSTRAINT_ERROR) is raised in the above cases.

T35. Check that a product or a quotient of fixed point values cannot be used in a comparison, a membership operation, or a real numeric type definition, e.g., $A*B = 6.0$ or $A * B = A * B$ are both illegal.

T36. Check fixed point multiplication and division when 'SMALL of the operands are not both powers of the same base value.

T37. Check that fixed point multiplication and division are not predefined when one of the operands is an integer type other than predefined INTEGER.

T38. Check that fixed point multiplication or division by a real literal is illegal.

Implementation Guideline: Use 0.0, 1.0, and other real literal values.

4.5.6 Highest Precedence Operators

Semantic Ramifications

S1. The type of $2 ** N$ or $3.14 ** N$ is determined by the context of the expression since the 2 (or the 3.14) must usually be implicitly converted to an appropriate type determined by the context; the conversion determines the type of the whole expression. Given more than one visible declaration of $**$, the call $**$ (2, N) or $**$ (3.14, N) is not by itself generally sufficient to determine which $**$ operator should be invoked. However,

$$2 ** N = 2 ** M$$

is legal even if more than one `****` is visible that takes a nonuniversal integer type as its first operand since equality is defined for *universal_integer*. Therefore, no implicit conversion is needed. (In this case, the *universal_integer* expression is nonstatic and must be computed at run time; see IG 4.10/S).

S2. `A ** B ** C` is forbidden syntactically, and the precedence rules imply that `-A ** B` is evaluated as `-(A**B)`.

Approved Interpretations

S3. When the RM requires that `NUMERIC_ERROR` be raised (other than by a raise statement), `CONSTRAINT_ERROR` can (and should) be raised instead (AI-00387).

Changes from July 1982

S4. `"abs"` and `"not"` now have the same precedence as `****`.

S5. `"not"` is predefined for any boolean type (and any one-dimensional array of any boolean type).

Changes from July 1980

S6. `"abs"` is no longer a function.

S7. The second operand of `****` must have the type predefined `INTEGER`.

Legality Rules

L1. The operand of predefined `"abs"` must be a numeric type (RM 4.5.6/1).

L2. The operand of predefined `"not"` must be a boolean type or a one-dimensional array type having a boolean component type (RM 4.5.6/1).

L3. The second operand of the predefined `****` operator must have the type predefined `INTEGER`; the first operand must have either an integer or a floating point type (RM 4.5.6/4).

Test Objectives and Design Guidelines

T1. Check that fixed point values cannot be exponentiated.

Check that floating or fixed point values cannot be used as the value of an exponent.

4.5.6.a Integer Exponentiating Operator

Semantic Ramifications

S1. Note that `X ** 4` can be evaluated as `(X*X) * (X*X)` although a similar rearrangement is not allowed for floating point (see IG 4.5.6.b/S).

Approved Interpretations

S2. When the RM requires that `NUMERIC_ERROR` be raised (other than by a raise statement), `CONSTRAINT_ERROR` can (and should) be raised instead (AI-00387).

Legality Rules

L1. The second operand of the predefined integer `****` operator must have the type predefined `INTEGER`; the first operand must have an integer type (RM 4.5.6/4).

Exception Conditions

- E1. **NUMERIC_ERROR** (or **CONSTRAINT_ERROR**; see AI-00387) is raised by predefined integer ******** if the result lies outside the base type (RM 4.5/7).
- E2. **CONSTRAINT_ERROR** is raised by predefined integer ******** if the second operand has a negative value (RM 4.5/6).

Test Objectives and Design Guidelines

- T11. Check that $X ** 0 = 1$ and $X ** 1 = X$ for all integer types.

Implementation Guideline: Use positive, negative, and zero values.

Implementation Guideline: Use named notation for some calls.

Check that exponentiation to a small integer value is correctly evaluated.

- T12. Check that exponentiation to large integer values is correctly evaluated.

- T13. Check that **NUMERIC_ERROR** (or **CONSTRAINT_ERROR**; see AI-00387) is raised if the expected value exceeds the range of the base type.

Implementation Guideline: Check both positive and negative values.

- T14. Check that **CONSTRAINT_ERROR** is raised if the exponent value is negative.

Implementation Guideline: Use both static and nonstatic exponent values.

- T15. Check that the exponent must not have any integer type other than predefined **INTEGER**.

4.5.6.b Floating Point Exponentiating Operator**Semantic Ramifications**

- S1. Since $0.0 ** 0 = 1.0$, it may be necessary to check explicitly for this case if, for instance, overflow is to be avoided.

- S2. For integer exponentiation, $X ** 4$ can be computed as $(X * X) * (X * X)$. Although any method can be used for floating point exponentiation as long as results are within the required model interval, there are cases where the error bounds for $X * X * X * X$ can be smaller than the bound for $(X ** 2) ** 2$. That is, the result of $(X ** 2) ** 2$ can lie outside the bounds for $X * X * X * X$ for some values of X . This means the only method guaranteed to stay within the error bounds specified by RM 4.5.7/9 is to perform $N-1$ multiplications when $N > 0$.

- S3. In practice, performing $N-1$ multiplications is not inefficient because 90 percent of the time the exponent is 2; 7 percent of the time, it is 3; and only in 3 percent of the cases is the exponent larger than three [Wichmann, B. A., "Algol 60 Compilation and Assessment," Academic Press, 1973]. So the possibility of reducing the number of multiplications does not arise very often.

- S4. $X ** N$ for negative N should not be computed as $(1.0/X) ** (\text{abs } N)$. For example, if X is 3.0 and N is 3, 81.0 is a model number and $1.0/81.0$ must lie within the smallest possible model interval. But $1.0/3.0$ is not a model number and so is not exactly representable as a binary floating point value. The accumulated error in multiplying an approximation to $1/3$ by itself will yield a result outside the required model interval if the exponent is sufficiently large.

Approved Interpretations

- S5. When the RM requires that **NUMERIC_ERROR** be raised (other than by a raise statement), **CONSTRAINT_ERROR** can (and should) be raised instead (AI-00387).

Legality Rules

- L1. The second operand of the predefined floating point ******** operator must have the type predefined **INTEGER**; the first operand must have a floating point type (RM 4.5.6/4).

Exception Conditions

- E1. **NUMERIC_ERROR** (or **CONSTRAINT_ERROR**; see AI-00387) is raised by predefined floating point ******** if the result lies outside the range of the operand's base type and **MACHINE_OVERFLOW**s is true for the base type (RM 4.5.7/7).
- E2. **NUMERIC_ERROR** (or **CONSTRAINT_ERROR**; see AI-00387) can be raised by predefined floating point ******** if the result lies outside the range of safe numbers for the operand type (RM 4.5.7/7).

Test Objectives and Design Guidelines

- T21. Check that ****** is performed correctly for a variety of floating point types, and in particular, that:
- a. $X ** 0 = 1.0$, $X ** 1 = X$, and $X ** (-1) = 1.0/X$.
 - b. Error bounds on the numerical result are acceptable.
 - c. Large variable exponent values are accepted.
 - d. Exponents of powers of two down to -30 are represented exactly.

Implementation Guideline: Use named notation for some calls.

- T22. Check that **NUMERIC_ERROR** (or **CONSTRAINT_ERROR**; see AI-00387) is raised if **MACHINE_OVERFLOW**s is true and the result is outside the range of the base type.
- T23. Check whether **NUMERIC_ERROR** (or **CONSTRAINT_ERROR**; see AI-00387) is raised if the result is outside the range of safe numbers, within the range of the base type, and **MACHINE_OVERFLOW**s is true.
- T24. Check whether **NUMERIC_ERROR** (or **CONSTRAINT_ERROR**; see AI-00387) is raised for floating point types if **MACHINE_OVERFLOW**s is false.
- T25. Check that the exponent cannot have any integer type other than predefined **INTEGER** when the base has a floating point type.

4.5.6.c Integer Absolute Value Operator**Semantic Ramifications****Approved Interpretations**

- S1. When the RM requires that **NUMERIC_ERROR** be raised (other than by a raise statement), **CONSTRAINT_ERROR** can (and should) be raised instead (AI-00387).

Legality Rules

- L1. The operand of predefined integer **abs** must have an integer type (RM 4.5.6/1).

Exception Conditions

- E1. **NUMERIC_ERROR** (or **CONSTRAINT_ERROR**; see AI-00387) is raised by predefined integer **abs** if the result lies outside the range of the base type (RM 4.5/7).

Test Objectives and Design Guidelines

T31. Check that `abs A` equals `A` if `A` is positive and equals `-A` if `A` is negative.

Implementation Guideline: Use named notation for some calls.

Implementation Guideline: Explicitly check zero as an argument value, among other cases. Check all predefined types.

T32. Check that `NUMERIC_ERROR` (or `CONSTRAINT_ERROR`; see AI-00387) is raised for `abs (T'BASE'FIRST)` if `-T'BASE'LAST > T'BASE'FIRST`.

4.5.6.d Real Absolute Value Operator (Fixed/Float)**Semantic Ramifications**

S1. For model numbers, the exact result is obtained since model values are symmetric with respect to the sign. With a fixed point type `F` filling a word on a two's-complement machine, the expression `-F'BASE'FIRST` will overflow and hence should raise `NUMERIC_ERROR` (or `CONSTRAINT_ERROR`; see AI-00387).

Approved Interpretations

S2. When the RM requires that `NUMERIC_ERROR` be raised (other than by a raise statement), `CONSTRAINT_ERROR` can (and should) be raised instead (AI-00387).

Legality Rules

L1. The operand of predefined floating point `abs` must have a floating point type (RM 4.5.6/1).

L2. The operand of predefined fixed point `abs` must have a fixed point type (RM 4.5.6/1).

Exception Conditions

E1. `NUMERIC_ERROR` (or `CONSTRAINT_ERROR`; see AI-00387) is raised by predefined floating point `abs` if the result lies outside the base type and `MACHINE_OVERFLOW` is true (RM 4.5.7/7).

E2. `NUMERIC_ERROR` (or `CONSTRAINT_ERROR`; see AI-00387) can be raised by predefined floating point `abs` if the result lies within the base type but outside the range of safe numbers (RM 4.5.7/7).

E3. `NUMERIC_ERROR` (or `CONSTRAINT_ERROR`; see AI-00387) is raised by predefined fixed point `abs` if the result lies outside the range of safe numbers (RM 4.5.7/7).

Test Objectives and Design Guidelines

T41. For a variety of floating point types, check that:

- a. for model numbers `A >= 0`, `abs A = A`,
- b. for model numbers `A <= 0`, `abs (A) = -A`,
- c. for operands that are not model numbers, the result is within the appropriate model interval.

Implementation Guideline: Use named notation for some calls.

T42. Check that `NUMERIC_ERROR` (or `CONSTRAINT_ERROR`; see AI-00387) is raised for floating point `abs` if `MACHINE_OVERFLOW` is true and the result lies outside the range of the base type.

Check whether `NUMERIC_ERROR` (or `CONSTRAINT_ERROR`; see AI-00387) is raised for floating point `abs` if `MACHINE_OVERFLOW` is false.

T51. For a variety of fixed point types, check that:

- a. for model numbers $A \geq 0$, $\text{abs } A = A$,
- b. for model numbers $A \leq 0$, $\text{abs } (A) = -A$,
- c. for operands that are not model numbers, the result is within the appropriate model interval.

Implementation Guideline: Use named notation for some calls.

T52. Check that `NUMERIC_ERROR` (or `CONSTRAINT_ERROR`; see AI-00387) is raised for fixed point types if the result lies outside the range of the base type.

4.5.6.e Scalar Negation Operator

Legality Rules

L1. The operand of predefined "not" for boolean types must be a boolean type (RM 4.5.6/1).

Test Objectives and Design Guidelines

T61. Check that nonboolean arguments are not allowed for the predefined not operator, including nonboolean scalar types and multidimensional arrays of boolean types.

T62. Check the truth table for not, including for derived boolean types.

Implementation Guideline: Use named notation for some calls.

4.5.6.f Array Negation Operator

Semantic Ramifications

S1. When "not" is applied to an array, the bounds of the result are the same as the bounds of the operand.

Legality Rules

L1. The operand of predefined "not" for arrays must have a one-dimensional array type with a boolean component type (RM 4.5.6/1).

Test Objectives and Design Guidelines

T71. Check that not is not predefined for multidimensional boolean arrays (see IG 4.5.6.e/T61).

T72. Check that not yields the correct results (both values and bounds) when applied to one-dimensional boolean arrays.

Implementation Guideline: Use named notation for some calls.

Implementation Guideline: Use packed arrays as well as arrays whose representation is chosen by the compiler.

Implementation Guideline: Use some arrays whose representation is unlikely to occupy a full word and some arrays whose representation is likely to occupy many words.

4.5.7 Accuracy of Operations with Real Operands

Semantic Ramifications

S1. This section specifies bounds on the accuracy with which floating and fixed point operations must be performed. The operations covered are "+", "-", "*", "/", "**", "abs", and conversions.

S2. For all practical purposes, machine operations conform to the requirements of this section, and hence, can be used directly by an implementation. There is one notable exception — machines that perform division by computing the reciprocal of the divisor and multiplying. Consider FIFTEEN/THREE, where the variables have the values implied by their names. Since 15.0, 3.0, and 5.0 are all model floating point numbers, an implementation is required to produce exactly the value 5.0 as the quotient of 15.0 and 3.0. However, if a machine actually computes $15.0/3.0$ as $15.0 * 0.333333...$, it is unlikely that a result exactly equal to 5.0 will be produced. In general, such a machine instruction cannot be used by a conforming Ada compiler.

S3. For floating point numbers, an implementation is permitted to raise `NUMERIC_ERROR` (or `CONSTRAINT_ERROR`; see AI-00387) as soon as a model interval associated with a result has a bound lying outside the range of safe numbers. However, if `MACHINE_OVERFLOW`s is true for floating point types, the RM requires that `NUMERIC_ERROR` (or `CONSTRAINT_ERROR`; see AI-00387) be raised if an attempt is made to produce a value that lies outside the range of the base type of the result. Neither exception need be raised if the result lies outside the range of safe numbers and within the range of the base type. In addition, RM 11.6/6 allows predefined numeric operations to be performed using a wider range than is otherwise required. `NUMERIC_ERROR` (or `CONSTRAINT_ERROR`) need not be raised in such a case if the correct result will be obtained, even if the result lies outside the range of the base type. For example, when evaluating $A*B/C$, the product might be held in a double-length register where it can be used directly as the dividend. No exception need be raised even if the product exceeds the range of the base type.

S4. If `MACHINE_OVERFLOW`s is false, `NUMERIC_ERROR` (or `CONSTRAINT_ERROR`) may nonetheless be raised for some floating point operations.

Approved Interpretations

S5. When the RM requires that `NUMERIC_ERROR` be raised (other than by a raise statement), `CONSTRAINT_ERROR` can (and should) be raised instead (AI-00387).

Changes from July 1982

S6. The conditions under which `NUMERIC_ERROR` can be raised have been changed.

Changes from July 1980

S7. The definitions are extended to include safe numbers.

S8. The model interval for the result of exponentiation is the interval defined for the underlying multiplication and division operations.

Test Objectives and Design Guidelines

T1. For floating point addition, subtraction, multiplication, division, and exponentiation, check whether `NUMERIC_ERROR` (or `CONSTRAINT_ERROR`; see AI-00387) is raised when a result is outside the range of safe numbers but within the range of the base type, and `MACHINE_OVERFLOW`s is true.

Implementation Guideline: Report whether `NUMERIC_ERROR` (or `CONSTRAINT_ERROR`) is raised even if `MACHINE_OVERFLOW`s is false.

4.6 Type Conversions

Semantic Ramifications

S1. Although `T("ABC")` is illegal (RM 4.6/3), `T("A" & "BC")` is allowed if only one catenation

operator is visible that takes arrays of characters as operands, since in this case, the type of the conversion operand is uniquely determined.

S2. A fixed or floating point operand value might lie outside the range of the target type before conversion but not after, e.g., `NATURAL (-0.4)` might yield the result zero. `CONSTRAINT_ERROR` is not raised in such cases.

S3. When converting a value to a fixed or floating point type, `NUMERIC_ERROR` (or `CONSTRAINT_ERROR`; see A1-00387) is raised if `MACHINE_OVERFLOW`s is true and the operand value lies outside the range of the target base type. `NUMERIC_ERROR` or `CONSTRAINT_ERROR` can also be raised if the operand value lies outside the range of safe numbers for the target type (whether or not `MACHINE_OVERFLOW`s is true). If the result lies outside the range of the target subtype, but within the range of safe numbers, `NUMERIC_ERROR` cannot be raised; `CONSTRAINT_ERROR` must be raised.

S4. For array conversions, when the RM says that index or component types must be the same, it means that the base types must be the same. Hence:

```
type T1 is array (NATURAL range <>) of INTEGER;
type T2 is array (INTEGER range <>) of NATURAL;
```

T1 and T2 are considered to have the same index and component types. An attempt to convert a value of type T1 to type T2 will, however, raise `CONSTRAINT_ERROR`, since the component subtype constraint for T1 is not the same as the constraint for T2.

S5. When converting null array values, it is possible for a non-null dimension of the operand to have values that lie outside the index subtype for the target array type:

```
subtype INT_8 is INTEGER range 1..8;
subtype INT_7 is INTEGER range 1..7;
type ARR_8_8 is array (INT_8 range <>, INT_8 range <>) of INTEGER;
type ARR_7_8 is array (INT_7 range <>, INT_8 range <>) of INTEGER;
...
... ARR_7_8 (ARR_8_8' (7..8 => (1..0 => 0))) )
```

In the conversion to `ARR_7_8`, the operand is a null array and the target type is an unconstrained array type. The non-null index range, 7..8, contains a bound, 8, that does not belong to the `INT_7` subtype. If no exception were raised by the conversion, the result would contain a non-null index range with a bound that lies outside its index subtype. It has been determined by the Language Maintenance Committee that the intent of the RM is that bounds of non-null index ranges should always belong to their index subtype. Therefore, when converting to an unconstrained array type, A1-00313 requires that `CONSTRAINT_ERROR` be raised if any bound of a non-null dimension of a conversion's operand does not belong to the corresponding index subtype of the target type.

S6. RM 4.6(15) gives a rule for interpreting expressions that contain literals, named numbers, and attributes returning values of type *universal_integer* or *universal_real*. The rule specifies, first, that if a legal interpretation exists when no implicit conversions are performed, this interpretation is used. This part of the rule allows expressions such as `ARR'LENGTH = 6` to be considered unambiguous -- the equality operator for *universal_integer* operands is used no matter how many equality operators are visible for other integer types.

S7. As another example, consider:

```
X := 1 + 2;
```

Suppose X is an `INTEGER` variable. The expression `1 + 2` can be interpreted without implicit conversions as yielding a *universal_integer* result, but such a result cannot be assigned to an

integer variable. In addition, the whole expression cannot be implicitly converted to INTEGER; only the literals can be implicitly converted. Since there is no legal interpretation of the assignment statement without implicit conversion, implicit conversions must be considered. The only implicit conversion that will succeed (i.e., that will allow the statement to be considered legal, assuming that only the predefined "+" operators are visible) will be the implicit conversion to INTEGER. So the expression is legal and invokes the INTEGER addition operator.

S8. Now consider the analysis of:

```
4**2 = 2**3
```

If only the predefined exponentiation operators are visible, there are no legal interpretations unless the exponents are implicitly converted to INTEGER. After these conversions, no further implicit conversions are needed to make the exponentiation legal, so the equality operator for *universal_integer* operands is used.

S9. Note that attributes are implicitly convertible. In particular, given

```
INTEGER' POS (1+2)
```

the 1, 2, and the attribute itself are all candidates for implicit conversion. Since POS accepts an argument having type *universal_integer*, no implicit conversions are applied to the literals 1 and 2.

S10. The attribute, CHARACTER'POS('A') is, technically speaking, parsed as a function call, since the syntax for attributes (RM 4.1.4/2) requires an argument that is a static expression having a universal type. The intent, however, is that implicit conversion is allowed for such an "attribute" (AI-00218).

S11. For array conversion, RM 4.6(11) requires that corresponding index types be convertible to each other. Index types must be discrete, i.e., an index type must be either an enumeration type or an integer type. Since there are no conversions defined between integer and enumeration types, the rule means that both index types must either be integer types or both must be enumeration types. Since any integer type is convertible to any other integer type (RM 4.6/7), there are no further requirements for integer index types. Enumeration base types are convertible only if they are identical or are related by derivation (RM 4.3/9), i.e., either one type is derived (directly or indirectly) from the other, or both types are derived from a common ancestor type.

S12. The RM does not specify what kind of rounding is to be done when converting half-integer real values to integer values. Most programmers would probably expect INTEGER(1.5) to equal 2, and INTEGER(-1.5) to equal -2. However, if a "round to even" rule is used when a value is exactly midway between two integer values, then INTEGER(1.5) = INTEGER(2.5) = 2, and for a two's-complement machine, INTEGER(-1.5) = -1 but INTEGER(-2.5) = -3. Another possibility is rounding up, in which case INTEGER(1.5) = 2 and INTEGER(-1.5) = -1; this is equivalent to adding 0.5 to the value and truncating the result to an integer value. Rounding down is also possible: INTEGER(1.5) = 1 and INTEGER(-1.5) = -2. Any of these possibilities is permitted by the RM. Moreover, the RM does not limit an implementation to just these methods; these are just the most likely methods to be supported by actual implementations.

S13. When the type mark in a conversion denotes a subtype that has less accuracy than its base type, the conversion is nonetheless performed with the accuracy of the base type. For example:

```
type T is digits 5;
subtype ST is T digits 3 range 12345.0 .. 15099.0;
... ST(12345.0) = T(12345.0)      -- must evaluate to TRUE
```

For subtype ST, 12345.0 falls in the model interval 12344.0 .. 12352.0 (since $12345.0 = 16\#3039.0\#$ and $ST_MANTISSA = 11$). Although 12345.0 is not a model number for subtype ST, the conversion, $ST(12345.0)$, is performed with 5-digit accuracy (AI-00407) and yields the result 12345.0. In addition, the bounds in the subtype declaration are evaluated using operations of the base type, and hence, are represented with the accuracy of the base type. (See IG 3.5.7/S for further discussion.)

Approved Interpretations

When the RM requires that `NUMERIC_ERROR` be raised (other than by a raise statement), `CONSTRAINT_ERROR` can (and should) be raised instead (AI-00387).

S14. When the target type in an array conversion is an unconstrained array type and the bounds of the operand do not lie within the range of the target type's index subtype, `CONSTRAINT_ERROR` can be raised instead of `NUMERIC_ERROR` (AI-00368).

S15. If a name can be considered either an attribute or a function call, the name is considered to be an attribute for purposes of deciding whether the value of the name can be implicitly converted to a numeric type (AI-00218).

Changes from July 1982

S16. The operand of a type conversion is not allowed to be the literal null, an allocator, an aggregate, or a string literal enclosed in parentheses.

Changes from July 1980

S17. A string literal is not allowed as the operand of a type conversion.

S18. Conversions between derived types are allowed if the operand and the target types are both derived from the same type.

S19. For array conversions, it is only required that corresponding index types be convertible to each other (rather than requiring that one be derived from the other).

S20. For array conversions, it is required that for component types with discriminants or for access component types, the component subtypes be both constrained or both unconstrained.

S21. `CONSTRAINT_ERROR` is raised for array conversions if any constraint on the component subtype is not the same for both target and operand types.

S22. For conversion to an unconstrained array type, `CONSTRAINT_ERROR` is raised (for non-null arrays) if any bound of the operand does not belong to the corresponding index subtype of the target type.

S23. For conversion to a constrained array subtype, `CONSTRAINT_ERROR` is raised if the lengths of corresponding dimensions do not match, or if only one array subtype is null.

S24. Literals are implicitly converted to some type instead of being considered overloaded.

Legality Rules

L1. The type of the operand of a type conversion must be determinable independently of the target type (RM 4.6/2).

L2. The operand of a type conversion must not be (RM 4.6/2):

- the literal null;
- an allocator;
- an aggregate;

- a string literal;
 - any of the above enclosed in one or more sets of parentheses.
- L3. If the target type is a numeric type, the operand must have a numeric type (RM 4.6/7).
- L4. If the target type is an array type, the type of the operand must satisfy the following conditions (RM 4.6/11):
- the operand type must also be an array type;
 - both types must have the same number of dimensions;
 - for each index position, the index types must either be the same or be convertible to each other, i.e.,
 - both index types must be integer types, or
 - both index types must be enumeration types, and
 - one type is derived from the other (directly or indirectly); or
 - both types are derived (directly or indirectly) from a common ancestor type;
 - the index types are the same.
 - the component base types must be the same;
 - if the component type is a type with discriminants or an access type, the component subtypes must be both constrained or both unconstrained.
- L5. If the target type is not an array type or a numeric type, then the operand base type must be (RM 4.6/9):
- the same as the target base type, or
 - derived (directly or indirectly) from the target type (or vice versa), or
 - there must exist a third type from which both types are derived (directly or indirectly).

Exception Conditions

- E1. **NUMERIC_ERROR** (or **CONSTRAINT_ERROR**; see AI-00387) is raised when converting to an integer type if the value of the operand lies outside the range of the target base type (RM 3.5.4/10 and RM 4.5/7).
- E2. **NUMERIC_ERROR** (or **CONSTRAINT_ERROR**; see AI-00387) is raised when converting to a fixed or floating point type if **MACHINE_OVERFLOW** is true and the value of the operand lies outside the range of the target base type (RM 4.5.7/7).
- E3. **NUMERIC_ERROR** (or **CONSTRAINT_ERROR**; see AI-00387) may be raised when converting to a fixed or floating point type if the value of the operand lies outside the range of safe numbers for the target type and inside the range of the target base type (RM 4.5.7/7).
- E4. **CONSTRAINT_ERROR** is raised for conversion to a numeric type or for conversion between derived types (other than array types; for array types, see E5) if the result of the conversion fails to satisfy a constraint imposed by the target type (RM 4.6/12). In particular:

- if the target type is an enumeration type, `CONSTRAINT_ERROR` is raised if the result of the conversion does not lie within the range of the target subtype.
- if the target type is a numeric type, `CONSTRAINT_ERROR` is raised if `NUMERIC_ERROR` is not raised and the result of the conversion does not lie within the range of the target subtype.
- if the target type is a record type with discriminants or a private type with discriminants, `CONSTRAINT_ERROR` is raised if the discriminant values for the operand do not equal those for the target type (RM 3.3/4 and RM 3.7.2/6).
- if the target type is a constrained access type whose designated type is an array type, `CONSTRAINT_ERROR` is raised if the operand value is not null and the index bounds of the designated array do not equal those of the target type (RM 3.3/4, RM 3.8/6, and RM 3.6.1/4).
- if the target type is a constrained access type whose designated type is a type with discriminants (a record type or a private type), `CONSTRAINT_ERROR` is raised if the operand value is not null and the discriminants of the designated object do not equal those of the target subtype (RM 3.3/4, RM 3.8/6, and RM 3.7.2/6).

E5. `CONSTRAINT_ERROR` is raised for conversion to an array type if:

- any (range, floating point, fixed point, index, or discriminant) constraint specified for the component subtype of the operand array type is not the same as the constraint specified for the target array's component subtype (RM 4.6/13).
- the target type is an unconstrained array type and a non-null index bound of the operand does not belong to the corresponding target type's index subtype (RM 4.6/13 and AI-00313).
- the target type is a constrained array type and if the target or operand type is non-null, corresponding dimensions do not have the same length (RM 4.6/13).

Test Objectives and Design Guidelines

Conversion of in out and out parameters is checked in IG 6.4.1/T3 and IG 6.4.1/T5.

- T1. Check that the type of the operand of an explicit conversion must be determined independently of the target type (see IG 8.7.b/T36).
- T2. Check that the operand of a type conversion must not be the literal null, an allocator, an aggregate, a string literal, or any of the preceding enclosed in one or more sets of parentheses.
Implementation Guideline: Be sure that there is one access type declared and be sure to use an aggregate that contains only character literals (i.e., that could have type `STRING`).
- T3. Check that when the target type is a numeric type, the operand type cannot be an enumeration type, an array type with a numeric component type, a record type with numeric components, an access type, or a private type whose full declaration declares a numeric type.
- T4. When the target type is an array type, check that:
- the operand type cannot be an enumeration type, a record type, an access type, or a private type whose full declaration declares an array type.

- the operand type cannot have fewer or more dimensions than the target type.

Implementation Guideline: Include a case where both types are null arrays.

- for corresponding index positions of the operand and target type, one index type cannot be an enumeration type and the other a numeric type; also, if both are different enumeration types, unless one is derived from the other or a common ancestor, the conversion is illegal.

- the component base type of an array operand type must be the same as the component base type for the target type.

Implementation Guideline: Include cases where component types are derived from a common ancestor and where both types are numeric.

- if the component subtype for the operand and target type is a record or private type with discriminants, the component subtype of the target cannot be constrained if the component subtype of the operand is unconstrained, and vice versa.

- if the component subtype for the operand and target type is an access type whose designated type is an array type or a type with discriminants, one component subtype cannot be constrained if the other is unconstrained.

- T5. Check that if the target type is an enumeration type, a record type, an access type, or a private type, the operand type cannot be a different enumeration type, record type, access type, or private type unrelated by derivation.

Integer Conversion

- T11. Check that integer conversions are performed correctly when the target and operand types are both integer types.

Implementation Guideline: The operand type should be another integer type unrelated to the target type by derivation, if possible.

Implementation Guideline: Include an identity conversion.

- T12. Check that integer conversions are performed correctly when the operand type is a floating-point type.

Implementation Guideline: Check for digits 5-29.

Implementation Guideline: Check that rounding is performed correctly, and report whether half-integer values are rounded up or down, using values such as -2.5, -1.5, 1.5, and 2.5.

- T13. Check that integer conversions are performed correctly when the operand type is a fixed-point type.

Implementation Guideline: Check for a variety of fixed-point mantissa lengths.

Implementation Guideline: Include, in separate tests, cases where TSMALL of the operand type is not a power of two.

Implementation Guideline: Check that rounding is performed correctly, and report whether half-integer values are rounded up or down, using values such as -2.5, -1.5, 1.5, and 2.5.

- T14. Check that NUMERIC_ERROR (or CONSTRAINT_ERROR; see AI-00387) is raised if the result of the conversion lies outside the range of the target type's base type.

Check that CONSTRAINT_ERROR is raised if the result of the conversion lies outside the range of the target type's subtype but within the range of the base type.

Implementation Guideline: Try a case such as NATURAL(-0.4) and NATURAL(-0.6). CONSTRAINT_ERROR need not be raised if the result of the conversion is zero.

Floating Point Conversion

- T21. Check that floating point conversions are performed correctly when the operand type is an integer value.

Implementation Guideline: Check for digits 5-29; use positive and negative values.

- T22. Check that floating point conversions are performed correctly for large *universal_integer* literals, and neither `NUMERIC_ERROR` nor `CONSTRAINT_ERROR` is raised by these conversions.

Implementation Guideline: Use digits 5-29 and the integer literals denoting positive and negative values of `LARGE`.

- T23. Check that floating point conversions are performed correctly when the operand type is a floating point type.

Implementation Guideline: Include conversions where the operand type has greater accuracy and the result is, or is not, a model number of the target type.

When the conversion is to a subtype having less accuracy than its base type, check that the result uses the model numbers of the base type (i.e., there must be no approximation using accuracy of the subtype, and the range check must be performed using the accuracy of the base type (see IG 3.5.7/T12)).

Check how floating point conversion discards excess bits: by rounding, by truncation (toward or away from zero), by "round to even," or in some other manner.

- T24. Check floating point conversions when the target type is a fixed point type.

Implementation Guideline: Include a set of tests where the fixed point model numbers are not powers of two.

- T25. Check that `NUMERIC_ERROR` (or `CONSTRAINT_ERROR`; see AI-00387) is raised if the result of a floating point conversion lies outside the range of the target type's base type and `MACHINE_OVERFLOW`s is true.

Implementation Guideline: Report whether `NUMERIC_ERROR` is raised even if `MACHINE_OVERFLOW`s is not true.

Check whether `NUMERIC_ERROR` (or `CONSTRAINT_ERROR`; see AI-00387) is raised when the result of a floating point conversion lies outside the range of safe numbers for the target type, but within the range of the base type.

Check that `CONSTRAINT_ERROR` is raised if the result of the conversion lies outside the range of the target subtype, but within the range of the base type.

Implementation Guideline: Include a case where the result lies outside the range of safe numbers, but within the range of the base type. `CONSTRAINT_ERROR` should be raised if and only if `NUMERIC_ERROR` is not raised.

Fixed Point Conversion

- T31. Check conversions to fixed point types when the operand type is an integer type.

- T32. Check conversions to fixed point types when the operand type is a floating point type.

Implementation Guideline: Check for digits 5-29.

- T33. Check conversions to fixed point types when the operand type is a fixed point type.

- T34. Check that `NUMERIC_ERROR` (or `CONSTRAINT_ERROR`; see AI-00387) is raised if the result of the conversion lies outside the range of the target type's base type.

Check that `CONSTRAINT_ERROR` is raised if the result of the conversion lies outside the range of the target subtype but within the range of the target base type.

Array Conversions

T41. Check array conversions when the target type is an unconstrained array type and the operand type requires conversion of index bounds.

Implementation Guideline: The target and operand types should not necessarily be related by derivation.

Implementation Guideline: Include cases where the representation of the operand and target index types is different (e.g., because one is LONG_INTEGER and the other is INTEGER).

T42. Check array conversions when the target type is a constrained array type and the operand type has bounds that do not belong to the target index type's base type.

Implementation Guideline: The bounds of the operand and target subtypes should be different.

Implementation Guideline: Include cases where the representation of the operand and the target types is different.

T43. Check that CONSTRAINT_ERROR is raised for conversion to an unconstrained array type if:

- component subtypes do not have the same constraints;

Implementation Guideline: Check for range, accuracy, index, and discriminant constraints. For index and discriminant constraints, check array, record, access, and private component types.

- for a non-null dimension of the operand type, one bound does not belong to the corresponding index subtype of the target type.

Implementation Guideline: Include a case where the target and operand type are both null arrays and corresponding dimensions do not have the same lengths.

T44. Check that CONSTRAINT_ERROR is raised for conversion to a constrained array type if:

- component subtypes do not have the same constraints;

Implementation Guideline: Check for range, accuracy, index, and discriminant constraints. For index and discriminant constraints, check array, record, access, and private component types.

- if the target type is non-null, corresponding dimensions of the target and operand do not have the same length.

- if the target type is null, the operand is non-null.

Derived Type Conversions

T51. Check that enumeration, record, access, private, and task values can be converted if the operand and target types are related by derivation.

Implementation Guideline: Include cases where the target and operand types are derived from a common ancestor, and cases where one is derived indirectly from the other.

Implementation Guideline: In separate tests, check that the operand and target types can have different representations (if the implementation allows the necessary representation clauses), and that the conversion produces the correct results.

Implementation Guideline: For record, access, and private types, include cases where the target subtype is constrained.

Implementation Guideline: Include cases where a record or private type is limited.

Implementation Guideline: Include a case where a type is derived from a generic formal type, e.g.:

```
generic
  type T is private;
package GP is
  type DT is new T;
end GP;
```

```
type ANCESTOR is ...;
type DA is new ANCESTOR;
type DB is new ANCESTOR;
```

```

package P is new GP(DB);
X : DA := ...;
Y : P.DT := P.DT(DA);      -- legal

```

- T52. Check that `CONSTRAINT_ERROR` is raised for conversion to an enumeration subtype if the converted value does not belong to the range of enumeration values for the target subtype.
- T53. Check that `CONSTRAINT_ERROR` is raised for conversion to a constrained record, private, or limited private subtype if the the discriminants of the target subtype do not equal those of the operand.
- T54. Check that `CONSTRAINT_ERROR` is raised for conversion to an access subtype if the operand value is not null, the designated subtype is an array type or a constrained type with discriminants, and the discriminants or index bounds of the designated object do not match those of the target subtype.

Check that `CONSTRAINT_ERROR` is not raised for conversion to an access subtype when the operand value is null.

4.7 Qualified Expressions

Semantic Ramifications

S1. When the operand in a conversion has the same base type as the target type, a conversion acts very much like a qualification. There are, however, a few differences. For arrays, the difference is that the bounds of the operand of a qualification must equal those of the type mark, whereas, in conversion, the operand's bounds can be different. In addition, for a conversion, the operand's type must be determined independent of the target type, whereas in qualification, the type mark determines the operand type if there is any ambiguity. Finally, the literal null, an allocator, an aggregate, and a string literal are all disallowed as operands of conversions, but are allowed as operands of qualified expressions.

S2. When the type mark in a qualification denotes a subtype that has less accuracy than its base type, the required range check is nonetheless performed with the accuracy of the base type. For example:

```

type T is digits 5;
subtype ST is T digits 3 range 12345.0 .. 15099.0;
... ST'(12345.0) = T(12345.0)      -- must evaluate to TRUE

```

For subtype ST, 12345.0 falls in the model interval 12344.0 .. 12352.0 (since 12345.0 = 16#3039.0# and ST'MANTISSA = 11). Although 12345.0 is not a model number for subtype ST, the qualification, ST'(12345.0), is evaluated with 5-digit accuracy (AI-00407) and yields the result 12345.0. In addition, the bounds in the subtype declaration are evaluated using operations of the base type, and hence, are represented with the accuracy of the base type. Consequently, no exception can be raised by the qualification. (See IG 3.5.7/S for further discussion.)

Changes from July 1982

S3. There are no significant changes.

Changes from July 1980

S4. There are no significant changes.

Legality Rules

- L1. The base type of the operand and the base type of the type mark must be the same (RM 4.7/3).

Exception Conditions

- E1. **CONSTRAINT_ERROR** is raised for a qualified expression if the value of the operand does not satisfy a constraint imposed by the type mark (RM 4.7/3). In particular:
- if the type mark is an enumeration or numeric type, **CONSTRAINT_ERROR** is raised if the value of the operand does not lie within the range of the target subtype.
 - if the type mark is a constrained array type, **CONSTRAINT_ERROR** is raised if the corresponding bounds of the type mark and the operand are not equal.
 - if the type mark is a record type with discriminants or a private type with discriminants, **CONSTRAINT_ERROR** is raised if the discriminant values for the operand do not equal those for the type mark.
 - if the type mark is a constrained access type whose designated type is an array type, **CONSTRAINT_ERROR** is raised if the operand value is not null and the index bounds of the designated array do not equal those of the type mark.
 - if the type mark is a constrained access type whose designated type is a type with discriminants (a record type or a private type), **CONSTRAINT_ERROR** is raised if the operand value is not null and the discriminants of the designated object do not equal those of the type mark.

Test Objectives and Design Guidelines

- T1. Check that the operand of a qualified expression must have the same base type as the type mark.

Implementation Guideline: In particular, check that operands that would be legal for conversions are illegal for qualifications.

- T2. Check that values belonging to each class of type can be written as the operand of a qualified expression.

Implementation Guideline: The type classes to be checked are: enumeration, integer, floating, fixed, array, record, access, private, limited private, task, and composite limited.

- T3. When the type mark denotes an enumeration type, check that **CONSTRAINT_ERROR** is raised when the value of the operand does not lie in the range of the type mark.

Implementation Guideline: Include a case where the type mark is a generic formal type.

- T4. When the type mark denotes an integer type, check that **CONSTRAINT_ERROR** is raised when the value of the operand does not lie in the range of the type mark.

- T5. When the type mark denotes a floating point type, check that **CONSTRAINT_ERROR** is raised when the value of the operand does not lie in the range of the type mark.

Check that when the type mark is a subtype with less accuracy than the base type, the range check is performed using the accuracy of the base type (see IG 3.5.7/T12).

- T6. When the type mark denotes a fixed point type, check that **CONSTRAINT_ERROR** is raised when the value of the operand does not lie in the range of the type mark.

Check that when the type mark is a subtype with less accuracy than the base type, the range check is performed using the accuracy of the base type (see IG 3.5.7/T12).

- T7. When the type mark denotes a constrained array type, check that `CONSTRAINT_ERROR` is raised when the bounds of the operand are not the same as the bounds of the type mark.

Implementation Guideline: Use both null and non-null arrays.

- T8. When the type mark denotes a constrained record, private, or limited private type, check that `CONSTRAINT_ERROR` is raised when the discriminants of the operand do not equal those of the type mark.
- T9. When the type mark denotes a constrained access type, check that `CONSTRAINT_ERROR` is raised when the value of the operand is not null and the designated object has index bounds or discriminant values that do not equal those specified in the access type's constraint.

Check that `CONSTRAINT_ERROR` is not raised when the operand value is null and the type mark denotes an access type.

4.8 Allocators

Semantic Ramifications

- S1. The type of an allocator is potentially ambiguous only if the type of the designated object appears in more than one access type definition, e.g.,

```
type PEOPLE is access PERSON;
type HUMAN is access PERSON;
```

The allocator, `new PERSON (...)`, can be either of type `PEOPLE` or `HUMAN`. Since in either case, an object of type `PERSON` is allocated, what are the implications of this type determination? There are two possibilities:

- if a collection size has been specified for `PEOPLE` that is smaller than the collection size for `HUMAN` (see RM 13.2), a `PEOPLE` allocation could raise `STORAGE_ERROR` when a `HUMAN` allocation would not;
- if `PEOPLE` values are represented with offset pointers (see IG 13.2.b/S) and `HUMAN` values with full pointers, then the representation of the run-time value depends on the type of the allocator.

- S2. Of course, knowing the possible types of `new PERSON (...)` is important in overloading resolution. Suppose we have

```
procedure F(X: HUMAN);
procedure F(X: DATA);
```

and `DATA` is not an access `PERSON` type. Then the call

```
F(new PERSON(...));
```

is clearly a call of the first procedure. Similarly, if `G` is an overloaded function such that `G.A` is either of type `HUMAN` or type `INTEGER`, then

```
G.A := new PERSON(M);
```

is resolved to invoke the first overloading of `G` and the access type is `HUMAN`. Finally, it should be noted that the expression:

```
new PERSON(...) = new PERSON (...)
```

would be illegal, since "=" is defined for operands of both type PEOPLE and type HUMAN. Even though the two allocators cannot possibly have the same value, a compiler cannot give the value FALSE for the above expression. The expression must be rejected as illegal because a unique "=" operator cannot be identified. (Of course, there is no ambiguity if there is only one visible "=" for an access PERSON type.)

S3. The type matching requirements of the various contexts in which an allocator can be used determine the type of the value returned by the allocator:

- in assignment statements, the type must be the same as the type of the left-hand side;
- in equality comparisons, the type must be the same as the type of the other operand;
- in component and object declarations, the type of the initial value must be the type of the declared component or object, respectively.
- in an aggregate, the type must be the component type required by the aggregate's type;
- as a default value in a parameter list, the type must be the type of the formal parameter;
- in qualifications, the type must be the same as the type used for qualification;
- as an expression in an allocator, e.g., new T'(new PERSON), the type of new PERSON must be the same as the base type of T;
- in a return statement, the type must be the type of the value to be returned;
- as an actual in parameter, the type must be that of the corresponding formal parameter;
- as an actual generic in parameter, the type must be that of the formal parameter.

See also IG 8.7/S.

S4. The base type of an allocator's type mark determines the type of the designated object, and hence, allocators need not use the same type mark as the one used in an access type declaration. Moreover, only the base type of the type mark can be used to resolve any ambiguity in the type of the allocator:

```

subtype MAN is PERSON(M);
subtype MALE is PERSON(M);
type ACC_MAN is access MAN;      -- MAN'BASE = PERSON
type ACC_MALE is access MALE;    -- MALE'BASE = PERSON
... new MAN ...

```

The allocator, new MAN, is either of type ACC_MAN or of type ACC_MALE; the fact that ACC_MAN was declared with the type mark MAN does not imply that new MAN has type ACC_MAN. All that is known is that an object of type PERSON has been allocated, and hence, new MAN has some access PERSON type. For example, if we add the following declarations:

```

V_MALE : ACC_MALE;
V_MAN  : ACC_MAN;

```

then

```

V_MALE := new MAN;
V_MAN  := new MALE;

```

are legal and unambiguously assign an ACC_MALE value to V_MAN and an ACC_MAN value to V_MAN.

s5. Similarly, given

```

type ACC_FEM is access PERSON(F);
OBJ_F : ACC_FEM;

```

then

```

if (new MAN) = OBJ_F then

```

is legal. However, the allocator raises CONSTRAINT_ERROR (since the designated object would have subtype PERSON(M), which does not satisfy the constraint imposed by the allocator's base type, ACC_FEM; AI-00397). It is not specified whether CONSTRAINT_ERROR is raised before or after creation of an object. In particular, default expressions for a created object could be evaluated before the exception is raised (AI-00397).

s6. In checking whether an allocator raises CONSTRAINT_ERROR, there are four sources of CONSTRAINT_ERROR to be considered:

1. a default initial value for a subcomponent of the designated object does not satisfy the corresponding subcomponent constraint.
2. the discriminant constraint or index constraint specified in an allocator is not compatible with the type mark given in the allocator, or the value specified as the operand of the qualified expression does not belong to the subtype denoted by the type mark.
3. the designated object does not belong to the designated subtype that is specified for the allocator's base type.
4. an index or discriminant subcomponent constraint that depends on a discriminant is not compatible with the subcomponent's type, and the subcomponent is present in the designated object.

s7. To illustrate the first two sources of CONSTRAINT_ERROR, consider the following declarations:

```

subtype INT1_10 is INTEGER range 1..10;
subtype FIVE is INTEGER range 5..5;

```

```

type A_INT4_6 is access INTEGER range 4..6;

```

```

OBJ : A_INT4_6 := new ...;      -- see allocators given below

```

The allocator new FIVE'(6) is legal and of type A_INT4_6. It raises CONSTRAINT_ERROR because the qualified expression, FIVE'(6), raises CONSTRAINT_ERROR. The allocator new INT1_10'(7) is also legal and of type A_INT4_6, but it raises CONSTRAINT_ERROR because 7 is not compatible with the range constraint associated with the allocator's base type, A_INT4_6 (AI-00397). Since none of the range constraints in the above example are required to be static expressions, in general an implementation will have to make at least two checks for constraint compatibility — one check for the qualified expression and the second for any constraint imposed in the allocator's base type definition.

s8. The following more complex example illustrates all four ways an allocator can raise `CONSTRAINT_ERROR`:

```

type GENDER is (M, F, UNKNOWN);
subtype GENDER_TYPE is GENDER range M..F;

type PERSON (GNDR: GENDER_TYPE := F) is
  record
    A : INTEGER range 1..10 := FUNC;
    B : TASK_TYPE;
  end record;

type PEOPLE is access PERSON;

type S_PERSON (GNDR: GENDER) is
  record
    X : PERSON (GNDR);
    Y : TASK_TYPE;
  end record;

type S_PEOPLE is access S_PERSON;

type A_PERSON_M is access PERSON(M);      -- case 1: access to cons-
                                           -- trained type
subtype PEOPLE_M is PEOPLE(M);           -- case 2: constrained access
                                           -- type
subtype PERSON_F is PERSON(F);

```

The various ways of raising `CONSTRAINT_ERROR` are:

1. A default initial value for a subcomponent of the designated object does not satisfy the subcomponent's constraint, e.g., `new PERSON(M)` when `FUNC` returns 11. (Of course, the evaluation of `FUNC` could itself cause an exception to be raised.)
2. The qualified expression or subtype indication with an explicit constraint raises `CONSTRAINT_ERROR`, e.g., `new FIVE'(6)` or `new PERSON (UNKNOWN)`.
3. The designated object does not satisfy a constraint specified for the access *base type*, e.g., `A_INT4_6'(new INTEGER'(7))` or `A_PERSON_M'(new PERSON (F))`. Note that `new PERSON (F)` is equivalent to `new PERSON`, i.e., `A_PERSON_M'(new PERSON)` will also raise `CONSTRAINT_ERROR`, because the default initial value for the discriminant is `F`.
4. A discriminant specified in the allocator is compatible with the discriminant's subtype, but is not compatible with its use in constraining a subcomponent, e.g., `new S_PERSON (GNDR => UNKNOWN)`.

s9. In the first of the four examples, `new PERSON(M)`, an object is created and then an exception may be raised when `FUNC` is invoked. If no exception is raised by `FUNC`'s invocation, `CONSTRAINT_ERROR` will be raised if `FUNC` returns a value outside the range 1..10. After component `A` is initialized, activation of the task component is attempted (RM 9.3/6). This activation might raise an exception (`PROGRAM_ERROR` or `TASKING_ERROR`).

s10. In case 4, the specified discriminant value (`UNKNOWN`) is compatible with `S_PERSON`'s discriminant specification but it is not compatible with the discriminant's use to constraint an

S_PERSON's X component (see RM 4.8/6 and RM 3.7.2/5). CONSTRAINT_ERROR is raised before (or after; AI-00397) the S_PERSON object is created, but before activation of the task component is attempted.

S11. There is a subtle difference between the third case, A_PERSON_M'(new PERSON(F)) and, PEOPLE_M'(new PERSON(F)). RM 3.8/3 says values of an access type designate objects whose subtype is that given in the access type definition. Hence

```
type A_PERSON_M is access PERSON(M);
```

is not equivalent to:

```
type %A_PERSON is access PERSON;
subtype A_PERSON_M is %A_PERSON(M); -- not equivalent to type decl.
```

since %A_PERSON values can designate objects of any PERSON subtype, but A_PERSON_M access values are only permitted to designate objects of type PERSON(M). In checking A_PERSON_M'(new PERSON(F)), the checks are performed as follows (the order is not significant):

1. Check the compatibility of the discriminant with the type mark given in the allocator. For this check PERSON(F) does not raise CONSTRAINT_ERROR, since F is compatible with PERSON's discriminant specification.
2. Check whether the subtype specified in the allocator (i.e., PERSON(F)) is consistent with the designated subtype specified in the declaration of the allocator's base type (i.e., A_PERSON_M). A_PERSON_M'(new ...) means that the allocator is required to produce an access value of type A_PERSON_M'BASE, i.e., an access value designating PERSON(M) objects. (An implementer needs to know what type of access value is produced by an allocator so the object can be allocated in the appropriate collection.)

Note that just as for the earlier INT1_10 examples, the constraint checks for an allocator involve both the checks imposed by the allocator's type mark *and* the checks imposed by the allocator's type. Both sets of constraints must be checked before or after creating an object.

S12. Note that from an implementation viewpoint, it is known that every object allocated in the A_PERSON_M collection *will be of the same size*. Note also that new PERSON(UNKNOWN) would raise CONSTRAINT_ERROR independently of the type of access value delivered by the allocator.

S13. In the fourth case, PEOPLE_M'(new PERSON(F)), the type produced by the allocator is PEOPLE_M'BASE, i.e., PEOPLE. Hence, a delivered access value will not violate any constraint of the access type, but it will violate the constraint imposed by the subtype PEOPLE_M. Since new PERSON(F) satisfies the constraint imposed by the access type PEOPLE_M'BASE, CONSTRAINT_ERROR is not raised prior to allocating the PERSON object in the collection PEOPLE. Hence, the PERSON object is allocated, and as a result of this allocation, FUNC is to be invoked and a task object is to be created and activated. If the value of FUNC satisfies the range constraint and the activation of the task component does not raise an exception, then an access value designating an object is delivered by the allocator. This value is checked against the constraint imposed by PEOPLE_M. Since the access value does not designate a PERSON(M) object, CONSTRAINT_ERROR is raised. The allocated object continues to exist at least as long as its task component is not terminated. Note that since FUNC need not return the same value for each call, and since activation of a task is not the same as invoking a function, RM 11.6 does not apply here: an object *must* be allocated, and the initializations attempted, even if a compiler can tell that an exception will be raised.

S14. Similar examples exist for array types:

```
type ARR is array (NATURAL range <>) of PERSON(M);
type ACC_ARR1_5 is access ARR(1..5);
type ACC_ARR is access ARR;
subtype ACC1_5_ARR is ACC_ARR(1..5);
```

They are:

1. new ARR(1..5) -- default initialization may raise an exception. Note that FUNC is called five times and five tasks are created and activated if no exceptions are raised;
2. new ARR(-5..-3) -- fails to satisfy NATURAL, the index subtype of ARR;
3. ACC_ARR1_5'(new ARR(2..6)) -- fails to satisfy the constraint on the allocator's base type; default initializations can be performed before the exception is raised (AI-00397).

S15. Ada is designed so that for a record or array type, an allocator need only allocate enough space to hold the particular subtype being allocated; no subsequent assignments are permitted to change the index or discriminant constraints associated with such an designated object.

S16. There is no requirement that the values of an allocator's index or discriminant constraint be static. In particular, discriminants governing variant parts need not be static.

S17. Note that for an access subtype, e.g.,

```
subtype FELLOW is PEOPLE(M);
```

an allocator new FELLOW is equivalent to new PEOPLE(M) (see RM 3.3.2/6); moreover, new FELLOW(M) is illegal, since a constrained type cannot be further constrained (RM 3.3.2/5).

S18. If a type T contains no components except a single discriminant, A, then new T(5) would be a legal allocator because the (5) is a discriminant constraint. new T'(5) would be illegal, since one-component aggregates must use named notation (see RM 4.3/4). However, new T'(A=>5) would be legal.

S19. Suppose a type contains two or more discriminants but no other components, e.g.,

```
type REC(A, B, C: INTEGER) is
  record
    null;
  end record;
```

```
subtype SREC is REC (3, 4, 5);
```

Now consider the following allocators:

```
new SREC (3, 4, 5);    -- illegal
new SREC' (3, 4, 5);  -- legal
```

The first allocator is illegal because (3, 4, 5) is a discriminant constraint, and SREC is already constrained. However, the second is a qualified expression, so the SREC allocator is legal. Note that if REC is a private type, then outside the package containing the full declaration of REC, new SREC'(3, 4, 5) will be illegal because there exist no aggregates for a private type (RM 7.4.2/4).

S20. Note that for arrays, an aggregate with the wrong number of components raises

CONSTRAINT_ERROR (see IG 4.3.2/S), but for records, a discriminant constraint or aggregate with the wrong number of components is illegal (see RM 4.3/6).

S21. Note that unconstrained access types may be constrained in allocators (see RM 3.8/6). Consider the following:

```

type ACC_PERSON is access PERSON;
type ACC_ACC_PERSON is access ACC_PERSON;
type ACC_MACC_PERSON is access ACC_PERSON(M);
X : ACC_ACC_PERSON := new ACC_PERSON(F);      -- (1)
XM : ACC_MACC_PERSON := new ACC_PERSON(F);    -- (2)
...
X.ALL := new PERSON(M);                      -- legal

```

The constraint in (1) has no effect on the allocated value (see last sentence RM 4.8/5); X.all has the value null until it is later assigned a different value. Note also that the first sentence of RM 4.8/5 does not apply to (1) since ACC_PERSON is not a type with discriminants (see RM 3.3/3); only record, private, and limited private types can have discriminants. The mere fact that a discriminant constraint can be given for an access type does not mean an access type has discriminants; the discriminant constraint is applied to the designated type (see RM 3.8/6). Note also that for (2), CONSTRAINT_ERROR is not raised, even though the specified discriminant value does not equal the value given in the declaration of ACC_MACC_PERSON. This is because the allocated value, null, satisfies any constraint, so no exception need be raised. CONSTRAINT_ERROR would be raised for the following allocator, however:

```

XM = new ACC_PERSON' (new PERSON(F));

```

The value of the innermost allocator is checked against the designated subtype for the outermost allocator. Since the outermost allocator has base type ACC_MACC_PERSON, the designated subtype is ACC_PERSON(M), and the subtype constraint is not satisfied by the value delivered by new PERSON(F). Hence CONSTRAINT_ERROR is raised.

S22. Finally, consider the following declarations:

```

type ACC_MPERSON is access PERSON(M);
type ACC_ACC_MPERSON is access ACC_MPERSON;
Z1 : ACC_ACC_MPERSON := new ACC_MPERSON(M);  -- illegal
Z2 : ACC_MACC_PERSON := new ACC_PERSON(M);   -- legal

```

The declaration of ACC_MPERSON imposes a constraint (on its designated objects). RM 3.3.2/5 says that a discriminant (or index) constraint cannot be imposed on a type mark that already imposes a constraint. Hence, ACC_MPERSON(M) is an illegal subtype indication. ACC_PERSON(M), however, is legal since ACC_PERSON does not impose a constraint on its designated objects.

S23. The RM requires that storage for a designated object not be reclaimed as long as the object is accessible directly or indirectly. There are at least two cases in which an object can become unnameable directly, but still be accessible in some sense:

1. storage for a task designated by an access value cannot be reclaimed before the task is terminated;
2. if a component of a designated object is renamed or passed as a parameter, storage for the designated object cannot be reclaimed as long as the component is accessible.

Each of these cases is illustrated by an example:

```

Case 1:
  declare
    task type T is ... end T;
    type T_ACC is access T;
    task body T is ... end T;
  begin
    declare
      LOST : T_ACC := new T;
      -- task is activated
    begin
      null;
    end;
    -- LOST is inaccessible but active

```

An implementation cannot reclaim the storage for LOST.all before LOST.all is terminated. Note also that LOST.all is not dependent on the inner block, so execution can leave the inner block before LOST.all terminates.

```

Case 2:
  declare
    type ARR is array (INTEGER range <>) of INTEGER;
    type ACC_ARR is access ARR;
    V_ARR : ACC_ARR := new ARR' (1..10 => 9);
    ELEM_10 : INTEGER renames V_ARR(10);
  begin
    V_ARR := null;           -- cannot reclaim ELEM_10 storage
    ELEM_10 := ELEM_10 + 1;   -- legal; no exceptions
    if ELEM_10 /= 10 then
      FAILED;
    end if;
  end;

```

Note that the same effect occurs when a component is passed as a generic in out parameter and may also occur for components passed (by reference) as subprogram parameters.

S24. Note that an implementer is not required to implement a garbage collector for access types. If no garbage collector is provided, the pragma CONTROLLED will have no possible effect. If garbage collection is provided, then an implementation has the option of obeying the pragma by turning off garbage collection for a specified type.

S25. Syntactically an allocator must conform to one of the following forms:

```

new T
new T X -- where X specifies a discriminant or range constraint.
new T'X -- where X specifies an aggregate or a parenthesized
        -- expression of type T.

```

The following table specifies various allocators and indicates their legality. The column headings have the following meanings. (Note: A type is a limited type if it is a limited private type, a task type, or a composite type with a component of a limited type.)

ST — scalar type

UR — unconstrained record type (with at least one discriminant)

CR — constrained record type (with or without discriminants)

- UP — unconstrained private type (with at least one discriminant)
 CP — constrained private type (with or without discriminants)
 UL — unconstrained limited type (with at least one discriminant)
 CL — constrained limited type (with or without discriminants)
 UA — unconstrained array type (component type is not a limited type)
 CA — constrained array type (component type is not a limited type)
 AT — access type

The row headings have the following meanings:

- (E,E) (E,E) has the form of a discriminant constraint
 (L..U) specifies one or more range constraints, including a type mark for a discrete type
 '(E) (E) is an expression
 '(A) (A) is an aggregate containing no **others** choice
 '(others) an aggregate containing an **others** choice, e.g., (**others** => 0)
 NULL X does not exist

The entries in the table refer to the notes following the table. An Xn indicates an illegal construct; the reason is given after the table. The upper left entry represents an allocator having the form new ST(E,E).

	ST	UR	CR	UP	CP	UL	CL	UA	CA	AT
(E,E)	X1	L1	X3	L1	X3	L1	X3	X1	X1	L1
(L..U)	X2	X2	X2	X2	X2	X2	X2	L2	X3	L2
'(E)	L3	L3	L3	L3	L3	X4	X4	L3	L3	L3
'(A)	X5	L4	L4	X5	X5	X5	X5	L4	L4	X5
'(others)	X5	L4	L4	X5	X5	X5	X5	X6	L4	X5
NULL	L5	L6	L5	L6	L5	L6	L5	X7	L5	L5

NOTES:

- L1 legal when E is a valid discriminant constraint for the type; note that CONSTRAINT_ERROR may be raised.
 L2 legal when (L..U) is a valid index constraint for the type.
 L3 legal when the expression is a legal expression for that type.
 L4 legal when the aggregate has the type expected.
 L5 legal; no initial values are required.
 L6 legal when a default value exists for the discriminant.
 X1 illegal; discriminant constraints cannot be applied to this type.

- X2 illegal; index constraints cannot be applied to this type.
- X3 illegal; constrained types cannot be further constrained.
- X4 illegal; no assignment for limited types (see RM 4.8/6 and RM 3.2.1/8).
- X5 illegal; aggregates cannot be written for this type.
- X6 illegal; an aggregate with others clause cannot be used with an unconstrained array type.
- X7 illegal; unconstrained arrays cannot be allocated.

Approved Interpretations

S26. `CONSTRAINT_ERROR` is raised if the object created by an allocator does not belong to the designated subtype of the allocator. This check is performed when evaluating the allocator but it is not further defined when the check is performed (AI-00397).

S27. Consider an allocator containing a discriminant constraint and whose type mark denotes an access type. `CONSTRAINT_ERROR` is raised by the allocator if and only if two conditions are met. First, each discriminant value belongs to the subtype of the corresponding discriminant of the designated type. Second, if the subtype defined by imposing the constraint on the designated type has components whose component subtype definitions depend on a discriminant, the corresponding discriminant value may (but need not) be substituted for the discriminant in each such component subtype definition; if this substitution is done, the compatibility of the resulting subtype indication is checked (AI-00007).

When an allocator has the form `new T` and `T` denotes a type with default discriminants, the initial values of the discriminants are given by the evaluation of the default expressions, and the corresponding constraint is then checked for compatibility (see IG 3.7.2/A1) (AI-00007).

Changes from July 1982

S28. There are no significant changes.

Changes from July 1980

S29. An allocator can only have the form `new subtype_indication` or `new qualified_expression`, i.e., the syntax of an allocator determines whether an initial value is being specified.

S30. The subtype indication can be an unconstrained record type with default discriminant values (e.g., `new PERSON`).

S31. The subtype indication can be an access type followed by a discriminant or index constraint.

S32. Initializations are performed as for declared objects. Qualified expressions provide explicit initializations, and initializations due to a subtype indication are implicit.

Legality Rules

L1. An allocator of the form `new T` is permitted only if (RM 4.8/4):

- `T` is a scalar or access type; or
- `T` is a constrained private, limited, record, or array subtype; or
- `T` has no discriminants and is a private type (limited or not), a task type or a record type; or
- `T` has discriminants with default values and is an unconstrained private or record type.

- L2. An allocator of the form `new T (E,E)`, where `(E,E)` has the form of a discriminant constraint is permitted only if (RM 4.8/4):
- T has discriminants, is an unconstrained record type or a private type, and `(E,E)` is a legal discriminant constraint for T; or
 - T is an unconstrained access type whose designated type is unconstrained and has discriminants, and `(E,E)` is a legal discriminant constraint for the designated type.
- L3. An allocator of the form `new T(L..U)`, where `(L..U)` has the form of an index constraint, is permitted only if T denotes an unconstrained array type or an access type whose designated type is an unconstrained array type, and the index constraint is legal for the designated type (RM 4.8/4).
- L4. An allocator of the form `new T'(E)`, where E is an expression, is permitted only if E has T's base type (RM 4.7/3) and T is not a limited type (RM 4.8/6 and RM 3.2.1/8).
- L5. An allocator of the form `new T'(A)`, where (A) is an aggregate having no others choice, is permitted only if T is an array type (constrained or unconstrained) or a record type (constrained or unconstrained) and (A) is a legal aggregate of type T (see RM 4.7/2 and RM 4.3).
- L6. An allocator of the form `new T constraint` is not permitted when the constraint has the form of a range constraint, floating point constraint, or fixed point constraint (RM 4.8/4).

Exception Conditions

- E1. For an allocator of the form `new T`:

- **CONSTRAINT_ERROR** is raised if the designated type for the allocator's base type is constrained:
 - T is a constrained record, private, or limited private type, and at least one of the discriminant values for T does not equal the corresponding value specified for the allocator's base type (AI-00397).
 - T is an unconstrained record, private, or limited private type with default discriminant values, and at least one default value for T does not equal the corresponding value specified for the allocator's base type (AI-00397).
 - T is a constrained array type, and at least one index bound in T does not equal the corresponding bound specified for the allocator's base type (AI-00397).
- **CONSTRAINT_ERROR** is raised if the initial value for a subcomponent of T does not satisfy the subcomponent's constraint (RM 4.8/6 and RM 3.2.1/16).
- **TASKING_ERROR** is raised if T has a subcomponent of a task type and if elaboration of the task body's declarative part causes an exception to be raised (RM 9.3/6-7).
- **PROGRAM_ERROR** is raised if T has a subcomponent of a task type and if the task body has not been elaborated when the allocator is evaluated (RM 3.9/6).

- E2. For an allocator of the form `new T X`:

- **CONSTRAINT_ERROR** is raised if:

- T is an unconstrained record, private, or limited private type, X is a discriminant constraint, and:
 - at least one of the values of X is outside the range of the corresponding discriminant (RM 3.3.2/9 and RM 3.7.2/5).
 - at least one of the values of X does not equal the corresponding value specified for the allocator's base type (AI-00397).
 - a subcomponent that exists for the subtype T X has a constraint that depends on a discriminant and the constraint is not compatible with the component's type (RM 3.7.2/5 and AI-00007; see also IG 3.7.2/E).
- T is an unconstrained array type, X is an index constraint, and:
 - at least one of the discrete ranges in X is not compatible with the corresponding index subtype (RM 3.6.1/4).
 - at least one index bound of X does not equal the corresponding bound specified for the allocator's base type (AI-00397).
- T is an unconstrained access type, X is a discriminant constraint, and at least one of the values of X is outside the range of the corresponding discriminant (RM 3.3.2/9 and RM 3.7.2/5). In addition, if the designated subtype has a subcomponent whose constraint depends on a discriminant, **CONSTRAINT_ERROR** may, but need not, be raised if the constraint is incompatible with the subcomponent's type (AI-00007).
- T is an unconstrained access type, X is an index constraint, and at least one of the discrete ranges in X is not compatible with the corresponding index subtype (RM 3.3.2/9 and RM 3.6.1/4).
- default initialization of at least one component of the allocated object does not satisfy the component's constraint (RM 4.8/6 and RM 3.2.1/16).
- **TASKING_ERROR** is raised if T has a subcomponent of a task type and if elaboration of the task body's declarative part causes an exception to be raised (RM 9.3/6-7).
- **PROGRAM_ERROR** is raised if T has a subcomponent of a task type and if the task body has not been elaborated when the allocator is evaluated (RM 3.9/6).

E3. For an allocator of the form `new T'(X)`, **CONSTRAINT_ERROR** is raised if:

- T is a scalar type and
 - X does not satisfy T's range constraint (RM 4.7/3).
 - X does not satisfy the range constraint imposed by the allocator's base type (AI-00397).
- T is a record or private type (constrained or unconstrained):
 - T is constrained, and:

- at least one of the discriminant values in X does not equal the corresponding discriminant constraint for T (RM 4.7/3).
- at least one of the discriminant values does not equal the corresponding value specified for the allocator's base type (AI-00397).
- T is unconstrained with discriminants:
 - at least one of the discriminant values in X does not satisfy the range constraint for the corresponding discriminant of T (RM 4.7/3).
 - the designated type for the allocator's base type is constrained (e.g., A_PERSON_M), and at least one of the discriminant values in X does not equal the corresponding value specified for the allocator's base type (AI-00397).
- T is an unconstrained array type, the designated type for the allocator's base type is constrained and at least one index bound does not equal the corresponding bound value specified for the allocator's base type (AI-00397).
- T is a constrained array type and
 - if X is using named notation, at least one of its (null or non-null) index bounds does not equal the corresponding bound specified for T (RM 4.7/3).
 - the number of components for a given dimension does not satisfy the index constraint for the corresponding index of T (RM 4.7/3)
 - the designated type for the allocator's base type is constrained and at least one index bound does not equal the corresponding bound value specified for the allocator's base type (AI-00397).
- T is an access type and
 - T is constrained, X is not null, and the value designated by X does not satisfy T's constraint (i.e., does not have the same bounds or discriminant values) (e.g., new FELLOW (new PERSON(F))) (RM 4.7/3).
 - T is unconstrained, T's base type is constrained, and the value designated by X does not satisfy the constraint specified for T's base type (e.g., new A_PERSON_M(new PERSON(F))) (AI-00397).
 - T is unconstrained, the allocator's base type is constrained, and the value designated by X does not satisfy the constraint specified for the allocator's base type (e.g., XM = new ACC_PERSON(new PERSON(F))) (AI-00397).

E4. STORAGE_ERROR is raised if insufficient storage remains for allocating objects of the specified type.

Test Objectives and Design Guidelines

Test objectives for allocators are distributed as follows, e.g., legal allocators having the form new T X are checked by objective 5.

	illegal	legal	exception
new T	1	4	7
new T X	2	5	8
new T' (X)	3		9

T1. Check that illegal forms of allocators are forbidden. In particular, for allocators of the form new T, check that:

- T cannot be an unconstrained record, private, or limited private type having discriminants without default values;

Implementation Guideline: Include a case where the allocator's base type specifies a constrained designated type.

- T cannot be an unconstrained array type.

Implementation Guideline: Include a case where the allocator's base type specifies a constrained designated type.

T2. Check that illegal forms of allocators are forbidden. In particular, for allocators of the form new T X, where X is a discriminant constraint or a value of type T enclosed in parentheses, check that:

- T cannot be a scalar type, a constrained record type, a constrained private type, a constrained limited type, or an array type (constrained or unconstrained);

Implementation Guideline: Check that X cannot have the form of an aggregate or a value of type T.

- T cannot be an access type whose designated type is a scalar type, a constrained record type, a constrained private type, a constrained limited type, an array type, or an access type (constrained or unconstrained);

Implementation Guideline: X should be a legal discriminant constraint for T's designated base type.

- T cannot be a constrained access subtype whose designated type has discriminants (e.g., PEOPLE_M);

Implementation Guideline: X should be a legal discriminant constraint for T's designated base type.

- if T is an unconstrained record, private, or limited private type with discriminants or an unconstrained access type whose designated type is an unconstrained record, private, or limited private type with discriminants, then X must be a legal discriminant constraint for T, namely:

- the discriminant_names given in the constraint cannot be different from the names of the discriminants of the type being constrained;
- the same name cannot appear twice as a discriminant_name in a particular discriminant_association or in different discriminant associations of the same discriminant constraint;
- if a mixture of named and positional association is used, a named discriminant association cannot give a value for a discriminant whose value has already been specified positionally;
- too many or too few discriminant values cannot be given.
- unnamed, (i.e., positional) discriminant values cannot be given after a discriminant association using discriminant names;

- the base type of the specified discriminant value cannot be different from the base type of the corresponding discriminant;
- **others** cannot be used as a discriminant name.

Implementation Guideline: For the **others** case, be sure all discriminants have the same type.

Check that for allocators of the form `new T X`, where `X` is an index constraint:

- `T` cannot be a scalar, record, private, or limited private type, nor can `T` be a constrained array type.
- `T` cannot be an access type whose designated type is a scalar, record, private, limited private, or access type, nor can the designated type be a constrained array type.
- `T` cannot be a constrained access subtype whose designated type is an array type.
- if `T` is an unconstrained array type or an unconstrained access type whose designated type is an unconstrained array type, then `X` must be a legal index constraint for `T`, namely:
 - the number of discrete ranges in the index constraint cannot be less than or greater than the number of indexes in the array type being constrained.
 - the base type of each discrete range in `X` cannot be different from the base type of the corresponding index.

T3. Check that illegal forms of allocators are forbidden. In particular, for allocators of the form `new T'(X)`, check that:

- if `T` is a scalar, access, or private type, `(X)` cannot be an aggregate, with or without **others**.
- if `T` is a record type and `(X)` is an aggregate, check all illegal forms of record aggregate.

Implementation Guideline: Check the following forms:

- choices with the same identifier;
- a choice given previously by a positional association;
- a choice that is not the name of a component;
- a named association with more than one choice when the corresponding components do not have the same type (but do have the same type class);
- a vacuous **others** choice.

- `T` cannot be a limited type (constrained or unconstrained);

Implementation Guideline: Check that:

- `X` cannot denote a value of a limited type;
Implementation Guideline: `T` should be a limited private type, a limited array type, a limited record type, and a task type.
- `X` cannot have the form of a positional aggregate whose values equal the corresponding discriminant values;
- `X` cannot have the form of a complete and valid aggregate for the underlying private type definition, including the case where the private type only has discriminant components;

Implementation Guideline: Try both array and record underlying types, including null arrays and null records.

- d. if T is an unconstrained array type, (X) cannot be an aggregate with an **others** choice.
- e. if T is an array type, check all forms of illegal array aggregate.

Implementation Guideline: The forms to be checked are:

- a positional component association preceding a named association that does not have the choice **others**.
- a choice that is a nonstatic expression or a static null discrete range in an aggregate with more than one component association or more than one choice.
Implementation Guideline: Try choices of the form E, L..R, ST, ST range L..R, and A'RANGE (where A is an array object or a constrained array type with static bounds and a static index subtype.)
Implementation Guideline: Include cases such as (F..G => 0), **others** => 1) and (2..1 => 0, **others** => 1) when the index subtype is static.
Implementation Guideline: In some cases, the index subtype should be a generic formal discrete type, or a type derived directly or indirectly from a formal type.
- an aggregate having more than one choice or component association, one choice is **others**, and the corresponding index subtype and discrete range (AI-00310) is not static.
Implementation Guideline: Include a use of a null static range and a vacuous **others** choice.
Implementation Guideline: Check where the component associations are both positional and named.
Implementation Guideline: Include a case where the subtype and index bounds for an index are static, but another dimension has either a nonstatic index subtype (with static index bounds) or nonstatic index bounds.
- for a non-null dimension of an aggregate, an index value between the lower and upper bound of the aggregate is not covered by the set of choice values.
Implementation Guideline: Include a null multidimensional aggregate with one non-null dimension, e.g.,


```
(1..2 => (2..1 => 1),
  -- 3 omitted
  4..5 => (2..1 => 2))
```
- an index value is represented more than once in the set of choices.
Implementation Guideline: Check for ranges that overlap, for duplicate choice values, and for an overlap between a choice value and a range.
- the type of a choice is not the same as the corresponding index type.
- the type of the expression specifying an array component value is not the same as the type of the array component.
Implementation Guideline: Include a case where the expression associated with a vacuous **others** choice is not the correct type.
- the innermost subaggregate of a multidimensional aggregate is enclosed in parentheses.
- an **others** choice is present when the type mark denotes an unconstrained array type.

T4. Check that the form new T is permitted if:

- T is a scalar subtype; or
- T is a constrained record, private, or limited private type; or
Implementation Guideline: Include a limited record type.
- T is an unconstrained record, private, or limited type whose discriminants have default values; or
Implementation Guideline: Check that only the appropriate default values for discriminants and other components are allocated; see IG 3.7.2/T8 for examples.
Implementation Guideline: Include a limited record type.
- T is a record, private, or limited type without discriminants; or
Implementation Guideline: Include a limited record type.
- T is a constrained array type.
Implementation Guideline: Check that the allocated object has the appropriate bounds.
Implementation Guideline: Include a limited array type.

- T is an access type.

Implementation Guideline: Check that CONSTRAINT_ERROR is not raised even if the allocator's base type specifies a constrained access type for its designated type, and a constraint imposed on T does not equal the constraint for the allocator's base type.

Implementation Guideline: Include a case where the designated type is an unconstrained array type or a type whose discriminants do not have defaults.

- T5. Check that an allocator of the form `new T X` allocates a new object each time it is executed, and that:

- if T is an unconstrained record, private, or limited type, the allocated object has the discriminant values specified by X;

Implementation Guideline: Check that default component values are properly set.

Implementation Guideline: Try one case where a discriminant governs a variant part and is nonstatic, including a case where a variant contains no components for one value of the discriminant, so the record consists only of discriminants for one set of discriminant values.

Implementation Guideline: Try one case where T has a single discriminant and X invokes an overloaded function returning a value of the type of the discriminant and a value of type T.

- if X is an index constraint and T an unconstrained array type, the allocated object has the index bounds specified by X.
- if T is an access type, T's designated type is unconstrained, and X is an appropriate index or discriminant constraint, check that the value null is allocated, and no CONSTRAINT_ERROR is raised even if the constraint does not satisfy a constraint specified for the allocator's base type (e.g., `XM = new ACC_PERSON(F)`).

- T6. Check that an allocator of the form `new T'(X)` allocates a new object each time it is executed, and that:

- if T is a scalar or access type, the allocated object has the value of X;
- if T is a record, array, or private type (constrained or unconstrained), the allocated object has the value of (X).

- T7. For allocators of the form `new T`, check that CONSTRAINT_ERROR is raised if:

- T is an unconstrained type with default discriminants (record, private, or limited) and one default discriminant value does not equal the corresponding value specified for the allocator's base type.
- T is an unconstrained type with default discriminants, a subcomponent constraint depends on a discriminant, the subcomponent is present in the designated object, and either a default discriminant value is not compatible with the subcomponent's type or a non-discriminant value in the subcomponent's constraint is not compatible with the subcomponent's type (see IG 3.7.2/T13, IG 3.7.2/T15, and IG 3.7.2/T16).
- T is a constrained type with discriminants (record, private, or limited) and at least one discriminant value specified for T does not equal the corresponding value specified for the allocator's base type.
- T is a constrained array type and at least one index bound for T does not equal the corresponding value specified for the allocator's base type.

Implementation Guideline: Include some null arrays and some multidimensional arrays.

- T8. For allocators of the form `new T X`, check that CONSTRAINT_ERROR is raised if:

- a. T is an unconstrained record, private, or limited type, X is a discriminant constraint, and

1. one of the values c is outside the range of the corresponding discriminant;
2. one of the discriminant values is not compatible with a constraint of a subcomponent in which it is used (see IG 3.7.2/T13, IG 3.7.2/T15, and IG 3.7.2/T16).

Implementation Guideline: If the subcomponent has an access type and is constrained by a discriminant constraint, the discriminant value should not belong to the subtype of the designated type's discriminant (AI-00007).

3. one of the discriminant values does not equal the corresponding value of the allocator's base type;
4. a default initialization raises an exception.

Implementation Guideline: Use the allocator as an actual parameter, in a qualification, and as an assigned value.

Implementation Guideline: Check that no default expressions are evaluated (or tasks activated) for cases 1, 2, and 3.

Implementation Guideline: Use both static and dynamic constraints.

- b. T is an unconstrained access type whose designated type is an unconstrained record, private, or limited private type with discriminants; X is a discriminant constraint; and one of the values of X is outside the range of the corresponding discriminant for the designated type;
- c. T is an unconstrained array type, X is an index constraint, and the bounds of X are not compatible with an index subtype of T;
- d. T is an unconstrained access type whose designated type is an unconstrained array type; X is an index constraint; and one of the non-null index bounds of X is outside the range of the corresponding index subtype for the designated type.

T9. For allocators of the form `new T'(X)`, check that `CONSTRAINT_ERROR` is raised if:

- a. T is a scalar subtype and X is outside the range of T, or is within T's range and outside the range of values permitted for objects designated by values of the allocator's base type (e.g., `A_INT4_6'(new INTEGER'(7))`).
- b. T is an unconstrained record or private type, (X) is an aggregate or a value of type T, and one of the discriminant values in X:

- does not satisfy the range constraint for the corresponding discriminant of T.
- does not equal the discriminant value specified in the declaration of the allocator's base type (i.e., the designated type is constrained in the base type's declaration).

Implementation Guideline: The value should satisfy the discriminant's range constraint.

- c. T is a constrained record or private type, (X) is an aggregate or a value of type T, and one of the discriminant values in X:
 - does not equal the corresponding discriminant value for T;
 - does not equal the corresponding discriminant value specified in the declaration of the allocator's base type.

Implementation Guideline: The value should satisfy T's constraint.

Implementation Guideline: Include a case where the designated type of the allocator is a constrained access type, e.g., access A_UR(2). Then give an allocator such as new A_UR'(new UR'(3, ...)).

d. T is an unconstrained array type with index subtype(s) S,

- X has too many values for S;

Implementation Guideline: The required number of values should be both statically and nonstatically defined.

- a named non-null bound of X lies outside S's range;
 - the bounds of X are not equal to the bounds specified for the allocator's designated base type. (They are equal to the bounds specified for T.)
- Implementation Guideline:* Try both null and non-null bounds.

e. T is a constrained array type and:

- a bound for X does not equal the corresponding bound for T;
 - a bound of T does not equal the corresponding value specified in the declaration of the allocator's base type.
- Implementation Guideline:* Both static and nonstatic constraints should be tried.
- a positional aggregate does not have the number of components required by T or by the allocator's base type.

f. T is a constrained or unconstrained multidimensional array type and all components of X do not have the same length or bounds.

Implementation Guideline: Include a case where bounds of a null array disagree.

g. T is a constrained access type and the object designated by X does not have discriminants or index bounds that equal the corresponding values for T.

Check that CONSTRAINT_ERROR is not raised if X is null.

h. T is an (unconstrained) access type, the designated type for T'BASE is a constrained access type, and the object designated by X does not have discriminants or index bounds that equal the corresponding values for T's designated type.

Implementation Guideline: For example, consider:

```
type UR (A : INTEGER) is ... ;
type A_CR is access UR(2);
type A_A_CR is access A_CR;
...
... new A_CR' (new UR(3));
```

i. T is an (unconstrained) access type whose designated type is a constrained access type, CA, and a discriminant or index value of X does not equal a value specified for CA.

Implementation Guideline: For example, consider:

```
type UR (A : INTEGER) is ... ;
type A_UR is access UR;
type A_CA_UR is access A_UR(2);
...
... new A_UR' (new UR(3));
```

j. T is an unconstrained access type, its designated type is also unconstrained,

and a discriminant value for X lies outside the range of the corresponding discriminant specification for the designated type, or a non-null index bound lies outside the range of an index subtype of the designated type.

T10. Check that null arrays and null records can be allocated.

Implementation Guideline: Compare the allocated object for equality; for null discrete arrays, the ordering operators can also be used.

T11. Check that overloaded allocators are determined to have the appropriate type.

Implementation Guideline: Check the use of equality tests, the assignment to a component of an overloaded function, and the calling of an overloaded subprogram with an access value.

Implementation Guideline: Use at least one allocator in which the type mark used in the allocator appears to resolve an ambiguity, but the base type is still ambiguous, e.g. new MAN.

T12. Check that discriminants governing variant parts need not be specified with static values in an allocator of the form new T X.

4.9 Static Expressions and Static Subtypes

Semantic Ramifications

s1. Several ambiguities in the wording of RM 4.9 have been clarified by approved interpretations. In particular, AI-00128 says that neither membership tests nor short-circuit control forms are allowed in static expressions because neither of these are operators.

s2. AI-00219 says further that the catenation operator and the IMAGE attribute are not allowed in static expressions because these operations yield nonscalar results:

```
'1' & '5' /= INTEGER'IMAGE(15)      -- not static
```

The above expression is not static even though '1', '5', and 15 are allowed in static expressions, INTEGER'IMAGE is an attribute of a static type, and /= produces a scalar result.

s3. AI-00251 says that a type derived (directly or indirectly) from a generic formal type is a nonstatic type. In addition, Commentary AI-00190 says that no static expression can have a generic formal type (or a type derived from a generic formal type). In particular, consider this example:

```
generic
  type T is range <>;
package PACK is
  type ARR is array (T) of BOOLEAN;
  X : ARR := (1_000_000 => FALSE,
              1_000_001 => TRUE);    -- illegal
end PACK;
```

RM 4.3.2/3 requires that when more than one component association is present in an array aggregate, all the choices must be static. The type of the choices in the above example is T. Since T is a generic formal type, the choices are not static (by AI-00190), and so the aggregate is illegal. On the other hand, the following aggregate is legal:

```
Y : ARR := (1_000_000..1_000_001 => TRUE);
```

When a single choice is used, it need not be static. Of course, the evaluation of the choice (at run time) will raise an exception if the values 1_000_000 and 1_000_001 do not both belong to subtype T. (NUMERIC_ERROR or CONSTRAINT_ERROR (see AI-00387) will be raised if either value does not belong to T's base type.)

S4. If PACK is instantiated with a static type, ARR will have a static index subtype in the instance (AI-00409):

```
subtype SMALL is INTEGER range 1..3;
package I is new PACK(SMALL);
X : I.ARR := (1 => FALSE, others => TRUE); -- legal
```

The aggregate is legal because I.ARR has a static index subtype, SMALL.

S5. If the evaluation of an expression would raise an exception, then the expression is considered as having no value, and is not static. Similarly, if an exception is raised while evaluating a subtype indication, the subtype indication is not considered static:

```
subtype INT is INTEGER range 1..5;
V : INTEGER range 1..6 := 3;
...
case V is
  when INT range 3..6 => ... -- illegal
```

An evaluation of the subtype indication INT range 3..6 would raise CONSTRAINT_ERROR. The choice, therefore, is not considered static, and since a choice in a case statement must be static (RM 5.4/5), the case statement is illegal.

S6. An expression whose value lies outside the range of its base type can be considered static even if its run-time evaluation would raise NUMERIC_ERROR. This can happen because a predefined operation may have a range wider than that of the base type of the operands (RM 11.6/6), in which case NUMERIC_ERROR need not be raised. So if the value of some subexpression lies outside the range of the base type, the containing expression can still be considered static as long as all the other rules are satisfied.

S7. If MACHINE_OVERFLOW is false, NUMERIC_ERROR (or CONSTRAINT_ERROR; see AI-00387) need not be raised for real expressions even if overflow occurs. However, a static expression must have a value, i.e., a value belonging to the expression's base type. If the model interval enclosing the result of an expression is undefined, in particular, if the computed value would lie outside the range of the base type, then the expression has no value. Hence, the expression cannot be considered static. In short, the value of MACHINE_OVERFLOW is irrelevant with respect to deciding whether a real expression is static.

S8. *Explicit* conversion is not allowed in a static expression, since conversion is not listed as a permitted primary. Since syntactic criteria are used to define which expressions are static (i.e., since only certain forms of primary are allowed), and since implicit conversions do not change the syntax of an expression, implicit conversions are allowed in static expressions. For example,

```
type NEW_INT is range 1..10;
V : NEW_INT := 3;
...
case V is
  when 5 => ... -- implicit conversion to NEW_INT
```

The existence of the implicit conversion does not make the choice illegal.

S9. If a renaming declaration declares a new name for a static constant (where a static constant is a constant declared by an object declaration with a static subtype and initialized with a static expression), then the new name can be used in a static expression (AI-00001), e.g.:

```

CONS  : constant INTEGER := 3;
RCONS : INTEGER renames CONS;
SCONS : INTEGER renames RCONS;

```

Both RCONS and SCONS can be used as a primary in a static expression, since both RCONS and SCONS denote CONS (see IG 8.5/S), and CONS is a constant that satisfies the rules for use in a static expression.

S10. If an enumeration literal is renamed as a function, the new name cannot be used in a static expression because the function name does not denote a predefined operator (RM 4.9/7). (Note that an enumeration literal is a predefined operation, not a predefined operator; see RM 3.3.3/2.) Similarly, if an attribute is renamed as a function, the new name cannot be used in a static expression.

S11. Certain attributes, PRED, SUCC, POS, and VAL, are considered functions. In particular, an attribute such as CHARACTER'POS('A') is, technically speaking, parsed as a function call, since the syntax for attributes (RM 4.1.4/2) requires an argument that is a static expression having a universal type. Such an attribute is intended to be considered static, even though it is a function call that does not denote a predefined operator.

S12. RM 4.9(6) only allows constants declared by object declarations to be used in static expressions. Other forms of constant are not allowed. The other forms of constant are: a loop parameter, a formal parameter of mode in (of a subprogram, entry, or generic unit), and a component of a constant array or record (RM 3.2.1/2).

S13. If the full declaration of a deferred constant declares a static constant, then within the scope of the full declaration, the constant can be used in static expressions. However, within the scope of the deferred constant declaration, the constant has a private type (not a scalar type), and so cannot be used in static expressions.

S14. The following contexts require static expressions:

- the initial value in a number declaration (RM 3.2/1);
- the bounds of an integer type definition (RM 3.5.4/3);
- the DIGITS value in a floating point constraint (RM 3.5.7/3);
- the bounds of a floating point constraint when the constraint is given in a type definition (RM 3.5.7/3);
- the value of DELTA given in a fixed point constraint (RM 3.5.9/3);
- the bounds of a fixed point constraint when the constraint is given in a type definition (RM 3.5.9/3);
- the argument given for the array attributes 'FIRST, 'LAST, 'RANGE, and 'LENGTH (RM 3.6.2/2);
- the choice in a variant part (RM 3.7.3/4);
- in an aggregate, the value specified for a discriminant that governs a variant part (RM 4.3.1/2);
- the choice in a case statement (RM 5.4/5);
- the specification of 'SIZE in a length clause (RM 13.2/5);
- the specification of 'SMALL in a length clause (RM 13.2/12);
- the choices and values specified in an enumeration representation clause (RM 13.3/4);

- the values specified in a record representation clause (RM 13.4/3).

In addition, the argument of the pragma `PRIORITY` is required to be static (RM 9.8/1); if the argument is not static, the pragma is not illegal, but is, instead, ignored (RM 2.8/11).

S15. Generally speaking, a compiler is required to evaluate a static expression at compile time when the context requires a static expression, since the value of the expression can affect the legality of a program. The compile-time evaluation of a floating point expression can produce a value different from what would be produced at run time if the evaluation is performed with different precision at compile and run time. In either case, of course, the value must fall within the required model interval.

S16. A renamed operator is allowed in a static expression if the new name is also an operator symbol (RM 4.9/7):

```
package P is
    type NEW_INT is range 1..100;
end P;

with P;
package RESTRICTED_P is
    subtype NEW_INT is P.NEW_INT;
    function "+" (L, R : NEW_INT) return NEW_INT renames P."+";
    function PLUS (L, R : NEW_INT) return NEW_INT renames P."+";
end RESTRICTED_P;
```

The package `RESTRICTED_P` declares a subset of the operators implicitly declared in package `P`. The newly declared `+` operator symbol can be used in a static expression having type `NEW_INT` since `RESTRICTED_P.+"` denotes the predefined operator `P.+"`:

```
with RESTRICTED_P; use RESTRICTED_P;
procedure EX is
    X : NEW_INT := 3 + 5;    -- uses RESTRICTED_P.+"
```

The function `PLUS` also denotes `P.+"`, but since `PLUS` is not an operator symbol, it cannot be used in a static expression:

```
RESTRICTED_P.PLUS(3, 5)    -- not static
RESTRICTED_P.+" (3, 5)    -- static
```

S17. The prefix of an array attribute must be either an object, a value, or an array subtype (RM 3.6.2/2 and RM 3.8.2/2). None of these prefixes are allowed for attributes used in static expressions. In particular, an array subtype prefix is not allowed because only scalar attributes can be static.

S18. The `VALUE` attribute cannot be used in a static expression because it returns a value that cannot have a scalar type (RM 3.5.5/12). The `IMAGE` attribute cannot be used in a static expression because it returns a nonscalar value (and because of A1-00128).

S19. The notion of a static index constraint is relevant in determining the static type of an aggregate that has an `others` choice (RM 4.3.2.3) and in determining the static type of a specification (RM 13.2/6) or a component clause (RM 13.4/3). The notion of a static discriminant constraint is only relevant in determining the static type of a component clause.

Approved Interpretations

S20. The use of a membership test in a static expression is not approved (AI-00128).

AD-A189 647

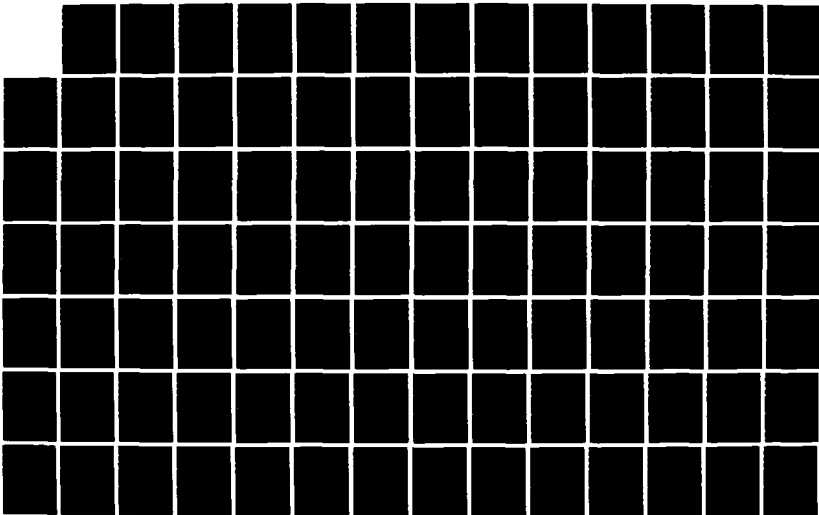
THE ADA (TRADE NAME) COMPILER VALIDATION CAPABILITY
IMPLEMENTERS' GUIDE VERSION 1(U) SOFTECH INC WALTHAM MA
J B GOODENOUGH DEC 86

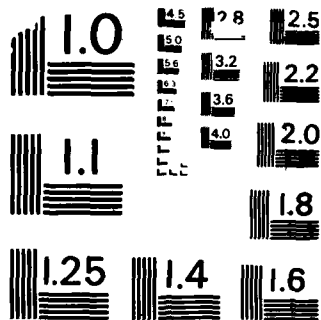
4/9

UNCLASSIFIED

F/G 12/3

NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

S21. In a static expression, every factor, term, simple expression, and relation must have a scalar type. (This implies that neither the concatenation operator nor the predefined attribute IMAGE can be used in static expressions.) (AI-00219)

S22. Types derived from generic formal types (directly or indirectly) are not static (AI-00251).

S23. A static expression is not allowed to have a generic formal type (including a type derived from a generic formal type, directly or indirectly) (AI-00190).

S24. If the name declared by a renaming declaration denotes a constant explicitly declared by a constant declaration having the form specified in RM 4.9/6, then the name can be used as a primary in a static expression (AI-00001).

S25. For a generic instantiation, if an actual generic parameter is a static subtype, then every use of the corresponding formal parameter within the instance is considered to denote a static subtype, even though the formal parameter does not denote a static subtype in the generic template (AI-00409).

Changes from July 1982

S26. Static function calls are required to have static actual parameters.

S27. Implementation-defined attributes of static subtypes are not considered static.

S28. Explicit type conversions are not allowed in static expressions.

S29. The definition of a static subtype is extended to include floating and fixed point constraints.

Changes from July 1980

S30. Expressions having composite types are not allowed in static expressions.

S31. Membership tests and short-circuit control forms are not allowed in static expressions.

S32. Components of arrays or records are not allowed in static expressions.

S33. A function call is allowed in a static expression if the function name is an operator symbol or an expanded name whose selector is an operator symbol (if the arguments of the function are also static expressions).

S34. A static expression enclosed in parentheses is allowed in a static expression.

S35. An expression is not static if its evaluation would raise an exception.

Legality Rules

L1. Every primary, factor, term, simple expression, and relation in a static expression must have a scalar type (RM 4.9/2). The type must not denote a generic formal type or a type derived (directly or indirectly) from a generic formal type (see IG 4.9/S).

L2. A static expression must not contain a membership test or a short circuit control form (RM 4.9/2 and AI-00128).

L3. The evaluation of a static expression must produce a value (i.e., an exception must not be raised) (RM 4.9/2). In particular,

- for static expressions having an integer type, the value must not lie outside the range of the base type, and the second operand of /, mod, or rem must not be zero.
- for static expressions having a fixed or floating point type, the value must not lie outside the range of the base type (whether or not MACHINE-_OVERFLOWS is true), and the second operand of / must not equal 0.0.

- for the evaluation of T'VAL, the argument must be in the range T'POS (T'BASE'FIRST) .. T'POS (T'BASE'LAST) (RM 3.5.5/7).
 - for the evaluation of T'SUCC, the argument must not equal T'BASE'LAST.
 - for the evaluation of T'PRED, the argument must not equal T'BASE'FIRST.
 - for the evaluation of a qualified expression, the operand value must belong to the range of the type mark (RM 4.7/3).
- L3. A static expression must not contain a name denoting a variable or a component of an array or record (RM 4.9/2-10).
- L4. A static expression must not contain an explicit type conversion (RM 4.9/2-10).
- L5. If a static expression contains the name of a constant, the name must denote a constant declared by a constant declaration with a static subtype, and initialized with a static expression (AI-00001).
- L6. A static expression must not call a user-defined function (RM 4.9/7).
- L7. If a static expression invokes a predefined operator, the function name must be an operator symbol or an expanded name whose selector is an operator symbol, and the actual parameters must all be static expressions (RM 4.9/7).
- L8. If a static expression contains an attribute, the attribute must not be 'ADDRESS, 'BASE, 'CALLABLE, 'CONSTRAINED, 'COUNT, 'FIRST (for arrays), 'FIRST(N), 'FIRST_BIT, 'IMAGE, 'LAST (for arrays), 'LAST(N), 'LAST_BIT, 'LENGTH, 'LENGTH(N), 'POSITION, 'RANGE, 'RANGE(N), 'SIZE (when the prefix denotes an object or a nonscalar type), 'STORAGE_SIZE, 'TERMINATED, or 'VALUE (RM 4.9/8).
- L9. If the attribute 'POS, 'VAL, 'SUCC, or 'PRED is used in a static expression, the attribute's prefix must be a static subtype and its argument must be a static expression (RM 4.9/8).
- L10. The following attributes can be used in a static expression if the attribute's prefix denotes a (scalar) static subtype: 'AFT, 'DELTA, 'DIGITS, 'EMAX, 'EPSILON, 'FIRST, 'FORE, 'LARGE, 'LAST, 'MACHINE_EMAX, 'MACHINE_EMIN, 'MACHINE_MANTISSA, 'MACHINE_OVERFLOW, 'MACHINE_RADIX, 'MACHINE_ROUNDS, 'MANTISSA, 'SAFE_EMAX, 'SAFE_LARGE, 'SAFE_SMALL, 'SIZE, 'SMALL, 'WIDTH (RM 4.9/8).
- L11. An implementation-defined attribute cannot be used in a static expression (RM 4.9/8).
- L12. If a static expression contains a qualified expression, the type mark must denote a static subtype and the operand must be a static expression (RM 4.9/9).
- L13. A static subtype (RM 4.9/11):
- must denote a scalar type;
 - must not denote a generic formal type or a type derived (directly or indirectly) from a generic formal type, nor may it denote a subtype of such a type.
 - must denote either a base type or a subtype defined by a subtype indication whose type mark, S, is a static scalar subtype and whose constraint is either a static range constraint or a floating or fixed point constraint whose range constraint, if any, is static. The constraint must be such that no exception would be raised, i.e., if the range constraint is non-null, both bounds must belong to S'FIRST .. S'LAST. In addition, for a floating point constraint, the specified digits value must not be greater than S'DIGITS; for a fixed point constraint, the specified value for delta must not be less than S'DELTA.

Test Objectives and Design Guidelines

- T1. Check that an expression having an array or record type is not considered static.
Implementation Guideline: Include an expression that contains a concatenation operator with scalar operands and a use of the IMAGE attribute for a static scalar type.

- T2. Check that no expression having a scalar generic formal type (or type derived indirectly from a generic formal type) is considered static.

Implementation Guideline: Use an array aggregate with an index type related to a generic formal type.

- T3. Check that a static expression is not allowed to contain a membership test or a short-circuit control form.

- T4. Check that a static expression must not raise an exception.

Implementation Guideline: Check for overflow, division by zero (for /, mod, and rem), a qualified expression, and the attributes 'VAL, 'SUCC, and 'PRED.

Check that the value of the argument to T'POS must lie within the range of T's base type (see IG 3.5.5/T4).

- T5. Check that a static expression must not contain a name denoting a variable or a component of an array or record.

Implementation Guideline: The array and record should be constants initialized with static values.

- T6. Check that a static expression must not contain an explicit type conversion.

- T7. Check that a static expression must not contain a name denoting a loop parameter, a subprogram in parameter, an entry in parameter, a generic formal in parameter, or a deferred constant (outside the scope of the deferred constant's full declaration).

Implementation Guideline: Include a name declared by a renaming declaration.

Implementation Guideline: The subtype of the objects should be static.

Implementation Guideline: Include the use of attributes of the object in an expression.

Check that a constant declared by an object declaration cannot be used in a static expression if the subtype used in the declaration was nonstatic, or if the constant was initialized with a nonstatic expression.

- T8. Check that a static expression cannot contain a call to a user-defined operator, nor can the name in the function call be an identifier, even if the identifier denotes a predefined operator.

Check that if an enumeration literal or static attribute is renamed as a function, the new name cannot be used in a static expression.

Check that the arguments of a function call cannot be nonstatic expressions if the function name is an operator symbol that denotes a predefined operator.

- T9. Check that the following attributes are not allowed in static expressions: 'ADDRESS, 'CALLABLE, 'CONSTRAINED, 'COUNT, 'FIRST (for an array prefix), 'FIRST(N), 'FIRST_BIT, 'IMAGE (see T1), 'LAST (for arrays), 'LAST(N), 'LAST_BIT, 'LENGTH, 'LENGTH(N), 'POSITION, 'RANGE, 'RANGE(N), 'SIZE (when the prefix denotes an object or a non-scalar type), 'STORAGE_SIZE, 'TERMINATED, and 'VALUE.

Implementation Guideline: Use those attributes that do not deliver scalar values in an equality operation, if possible.

Check that a static expression cannot contain the attributes 'POS, 'VAL, 'SUCC, or 'PRED if the prefix of these attributes denotes a nonstatic subtype, or if the argument is a nonstatic expression.

Check that a static expression cannot contain one of the following attributes if its prefix

denotes a nonstatic subtype: 'AFT, 'DELTA, 'DIGITS, 'EMAX, 'EPSILON, 'FIRST, 'FORE, 'LARGE, 'LAST, 'MACHINE_EMAX, 'MACHINE_EMIN, 'MACHINE_MANTISSA, 'MACHINE_OVERFLOW, 'MACHINE_RADIX, 'MACHINE_ROUNDS, 'MANTISSA, 'SAFE_EMAX, 'SAFE_LARGE, 'SAFE_SMALL, 'SIZE, 'SMALL, 'WIDTH.

Implementation Guideline: Include a prefix that denotes a generic formal integer or discrete type, or a type derived directly or indirectly from such a type.

- T10. Check that a static expression cannot contain a qualified expression if the type mark denotes a nonstatic type (scalar or not), or the argument is a nonstatic scalar expression.
- T11. Check that a static subtype cannot be a generic formal type, a subtype of a generic formal type, or a type derived indirectly from a generic formal type.

Implementation Guideline: Include the use of a generic formal type and a type indirectly derived from a generic formal type. Use the type in a qualified expression, a prefix of an attribute, and in a constant declaration.

Check that a subtype indication is nonstatic if the type mark is nonstatic.

Implementation Guideline: The range should be specified with static expressions.

Check that a subtype indication is nonstatic if its evaluation would raise an exception.

Implementation Guideline: Include exceptions raised for range checks, for incorrect digits values, and for incorrect values of delta.

- T20. Check that enumeration literals (including character literals) can be used in static expressions together with relational and equality operators.
- T21. Check that boolean literals can be used in static expressions together with the logical operators, the operator not, and the relational and equality operators.
- T22. Check that numeric literals and named numbers can be used in static expressions together with the unary operators + and -, the abs operator, the binary + and - operators, the *, /, **, mod, and rem operators, and the relational and equality operators.

Implementation Guideline: Include parenthesized static expressions.

- T23. Check that a constant declared by an object declaration can be used in a static expression if the constant was declared with a static subtype and initialized with a static expression.

Implementation Guideline: Check constants having an enumeration, integer, floating point, and fixed point type.

Implementation Guideline: Include a constant declaration that is the full declaration of a deferred constant.

Check that a renamed static constant can be used in a static expression.

- T24. Check that a function call can appear in a static expression if the function name denotes a predefined operator and has the form of an operator symbol or an expanded name whose selector is an operator symbol.

Implementation Guideline: Include the use of a renamed predefined operator.

- T25. Check that the following attributes can be used in a static expression: 'SUCC, 'PRED, 'POS, 'VAL, 'AFT, 'DELTA, 'DIGITS, 'EMAX, 'EPSILON, 'FIRST, 'FORE, 'LARGE, 'LAST, 'MACHINE_EMAX, 'MACHINE_EMIN, 'MACHINE_MANTISSA, 'MACHINE_OVERFLOW, 'MACHINE_RADIX, 'MACHINE_ROUNDS, 'MANTISSA, 'SAFE_EMAX, 'SAFE_LARGE, 'SAFE_SMALL, 'SIZE, 'SMALL, 'WIDTH.

Check that T'BASE is a static subtype if T denotes a scalar type other than a generic formal type or a type derived (directly or indirectly) from a generic formal type.

Implementation Guideline: T should denote a nonstatic subtype. Use T'BASE as the prefix for each allowable static attribute.

Check that T'BASE denotes a nonstatic subtype type if T is a scalar generic formal type or is derived (directly or indirectly) from a scalar generic formal type.

T26. Check that a qualified expression can appear in a static expression.

Implementation Guideline: Check for enumeration, integer, floating point, and fixed point types.

4.10 Universal Expressions

Semantic Ramifications

S1. To correctly evaluate static *universal_real* expressions, an implementation in general must use rational arithmetic. For example, consider:

```
case B is
  when (0.1 * 0.1 = 0.01) => ...
  when FALSE => ...
end case;
```

This statement is legal only if the first choice evaluates to TRUE. If a binary approximation to 0.1 is used, then it is unlikely that $0.1 * 0.1$ will evaluate as being equal to 0.01. However, it is required that every implementation evaluate $0.1 * 0.1$ exactly (RM 4.10/4), producing TRUE as the result of the equality operation.

S2. Similarly, consider whether the following declarations conform:

```
procedure P (X : FLOAT := 3#0.1#);
procedure P (X : FLOAT := 0.3333333333333333) is ...;
```

Since there is no exact decimal representation for the value $1/3$, the literals do not have the same value, and so these two specifications do not conform (RM 6.3.1/2).

S3. Relational and membership operations for static *universal_real* operands must be evaluated exactly. The RM defines the accuracy of relational and membership operations in terms of model numbers of the type (see RM 4.5.7/10, /11). According to RM 3.5.6/3, a set of model numbers is associated with every real type, and RM 3.5.6/5 says *universal_real* is a real type, so there are model numbers for the type *universal_real*. The RM does not specify the form of the model numbers (e.g., do they have the form specified in RM 3.5.7/4 with infinite mantissa, or the form given RM by 3.5.9/4 with an appropriate 'SMALL?'), but regardless of the form that is considered to be used, it is clear from RM 4.10/4 that the model interval for a static *universal_real* expression is a point, and so relational and membership operations for static *universal_real* operands are to be evaluated exactly, as required by RM 4.5.7/10 when comparing model numbers of a type.

S4. Suppose the operands of a relational operator or membership test have the type *universal_real* and one or more of the operands is nonstatic. The static operands must be evaluated exactly, since RM 4.10/4 says that "if a universal expression is a static expression, then the evaluation must be exact." Consider a relational expression

NS relation S

where NS is a nonstatic *universal_real* expression and S is a static *universal_real* expression. Since the values of S and NS, being of type *universal_real* are both model numbers, the relation itself must be evaluated exactly using the computed values of S and NS. It might at first seem that the expression S must be carried to full precision (i.e., as a ratio of arbitrarily large integers) at run time. But this is not, in fact, the case.

S5. Let LONG be the base type for the highest precision floating point numbers used by a given implementation. By abuse of notation, we shall also use it to denote the set of all values of type LONG. Let CEIL(x, LONG) be the least upper bound of the subset of LONG greater

than or equal to x . Let $\text{FLOOR}(x, \text{LONG})$ be the greatest lower bound of the subset of LONG less than or equal to x . These may be undefined beyond the extrema of LONG . There are two cases, and the one that applies can be determined at compilation time.

1. The value of S is a member of LONG . In this case, the implementation is obvious.
2. S is not a member of LONG . Convert the relational expression according to the following table:

Expression	Transformed expression
$\text{NS} > S, \text{NS} \geq S,$ $S < \text{NS}, S \leq \text{NS}$	$\text{NS} > \text{FLOOR}(S, \text{LONG}),$ if the latter is defined or TRUE otherwise
$\text{NS} < S, \text{NS} \leq S$ $S > \text{NS}, S \geq \text{NS}$	$\text{NS} < \text{CEIL}(S, \text{LONG}),$ if the latter is defined or TRUE otherwise
$\text{NS} = S, S = \text{NS}$	FALSE
$\text{NS} \neq S, S \neq \text{NS}$	TRUE

We thus reduce everything to at worst the case of comparing a static member of LONG to a nonstatic member of LONG . As a consequence, it is possible to maintain the convenient "semantic fiction" that S is carried to infinite precision in the comparison without run time cost.

S6. The above arguments apply equally well when the relation is a membership test, since X in $L..R$ is evaluated as $X \geq L$ and $X \leq R$.

S7. Nonstatic expressions having a universal type can be created in several ways, and, in fact, any nonstatic integer value can be converted to a nonstatic *universal_integer* or *universal_real* value, and vice versa:

```

procedure P (V : STRING) is
begin
    ... V'LENGTH                -- nonstatic universal_integer
    ... 3**N                    -- nonstatic universal_integer
    ... T'POS(INTEGER(F))       -- conversion to universal_integer
    ... 3.0**N                  -- nonstatic universal_real
    ... 1.0*T'POS(N)            -- conversion to universal_real
end;
```

S8. RM 4.10/5 says, in essence, that nonstatic *universal_integer* expressions can be evaluated with the largest predefined integer type. RM 4.10/4 allows nonstatic *universal_real* expressions to be evaluated using the most accurate predefined floating point type. With respect to *universal_real* expressions, this means results computed at run time can be significantly different from those that would be computed for a static expression having the same operand values.

S9. Since a static expression cannot raise an exception (RM 4.9/2), division by zero, for example, does not make a universal expression illegal, but instead, makes it nonstatic. **NUMERIC_ERROR** (or **CONSTRAINT_ERROR**; see AI-00387) must then be raised at run time:

```

X : BOOLEAN := 1 < 1/0;          -- NUMERIC_ERROR is raised
```

S10. When evaluating nonstatic expressions having a universal type, **NUMERIC_ERROR** (or **CONSTRAINT_ERROR**; see AI-00387) can be raised not merely when the result lies outside a certain range, but also if an *operand* has a value outside that range (AI-00181). For example, consider:

10E1000 mod SYSTEM.MAX_INT

The left operand can readily be made to exceed SYSTEM.MAX_INT, but the result will always be less than SYSTEM.MAX_INT. Since the right operand value can be a nonstatic expression, run-time evaluation of the left operand must be allowed to raise NUMERIC_ERROR (or CONSTRAINT_ERROR; see AI-00387). Similar examples can be constructed for nonstatic *universal_real* expressions. The effect of AI-00181 is to ensure that it is always possible to use a predefined type when computing nonstatic universal values.

Approved Interpretations

S11. For the evaluation of an operation of a nonstatic universal expression, an implementation is allowed to raise the exception NUMERIC_ERROR if any operand or the result is a real value whose absolute value exceeds the largest safe number of the most accurate predefined floating point type (excluding *universal_real*), or an integer value greater than SYSTEM.MAX_INT or less than SYSTEM.MIN_INT (AI-00405).

S12. When the RM requires that NUMERIC_ERROR be raised (other than by a raise statement), CONSTRAINT_ERROR should be raised instead (AI-00387).

Changes from July 1982

S13. It is allowed to raise NUMERIC_ERROR only when evaluating a nonstatic *universal_integer* expression.

S14. It has been clarified that evaluation of nonstatic *universal_real* expressions should use the most accurate predefined floating point type.

S15. The condition for raising NUMERIC_ERROR for *universal_real* is restated to use the most accurate predefined floating point type (instead of any predefined floating point type).

Changes from July 1980

S16. Nonstatic *universal_integer* and *universal_real* expressions are now possible.

Legality Rules

- L1. If one operand of a relational or equality operator has the type *universal_integer*, the type of the other operand must not be *universal_real*, and vice versa (RM 4.10/2 and RM 4.5.2/1).
- L2. For a membership test of the form E in L..R or E not in L..R, if two of the expressions E, L, and R have the type *universal_integer*, the third must not have type *universal_real*, and vice versa (RM 4.10/2 and RM 4.5.2/10).
- L3. If one operand of binary "+" or binary "-" has the type *universal_integer*, the other operand must not have type *universal_real*, and vice versa (RM 4.10/2 and RM 4.5.3/2).
- L4. Neither operand of rem and mod is allowed to have type *universal_real* (RM 4.10/2 and RM 4.5.5/1).
- L5. If the first operand of the division operator has type *universal_integer*, the second operand must not have type *universal_real* (RM 4.10/3, RM 4.10/2, and RM 4.5.5/1).
- L6. If the first operand of the exponentiation operator has type *universal_integer* or *universal_real*, the second operand must have type predefined INTEGER (RM 4.10/2 and RM 4.5.6/4).

Exception Conditions

- E1. NUMERIC_ERROR (or CONSTRAINT_ERROR; see AI-00387) is raised when evaluating

a nonstatic expression having type *universal_integer* if the value of an operand or the result of the expression lies outside the range `SYSTEM.MIN_INT .. SYSTEM.MAX_INT` (RM 4.10/5 and AI-00181).

- E2. `NUMERIC_ERROR` (or `CONSTRAINT_ERROR`; see AI-00387) is raised for the `/`, `mod`, and `rem` operators if the type of both operands is *universal_integer*, the value of the divisor is zero, and the expression is not required to be static (RM 4.10/5).
- E3. `CONSTRAINT_ERROR` is raised for an exponentiation operator if the first operand has type *universal_integer*, if the value of the second operand is negative, and if the expression containing the operator is not required to be static (RM 4.10/5 and RM 4.5.6/6).
- E4. `NUMERIC_ERROR` (or `CONSTRAINT_ERROR`; see AI-00387) is raised when evaluating a nonstatic expression having type *universal_real* if the value of the expression lies outside the range of the base type of the most accurate predefined floating point type and `MACHINE_OVERFLOW` is true for this type (RM 4.10/5 and RM 4.5.7/7).
- E5. `NUMERIC_ERROR` (or `CONSTRAINT_ERROR`; see AI-00387) may be raised when evaluating a nonstatic expression having type *universal_real* if the value of the expression is within the range of the base type of the most accurate predefined floating point type but is outside the range of the type's safe numbers (RM 4.10/5 and RM 4.5.7/7).
- E6. `NUMERIC_ERROR` (or `CONSTRAINT_ERROR`; see AI-00387) is raised for the division operator if the first operand has type *universal_real*, the value of the divisor is zero, the type of the divisor is *universal_integer* or *universal_real*, and the expression is not required to be static (RM 4.10/5 and RM 4.5.7/7).

Test Objectives and Design Guidelines

- T2. Check that static *universal_integer* expressions are evaluated correctly when large literal values are used, but the result is a small integer value.
- T4. Check that static *universal_integer* expressions are evaluated correctly when large literal values are used and the result is a large value.
- T5. For *universal_integer* expressions, check that `NUMERIC_ERROR` (or `CONSTRAINT_ERROR`; see AI-00387) is raised:
 - if division by zero is attempted.
 - if the second operand of `rem` or `mod` is zero.
 - for nonstatic expressions, if an operand value or the result lies outside the range `SYSTEM.MIN_INT .. SYSTEM.MAX_INT`.
- T6. Check that `CONSTRAINT_ERROR` is raised for a *universal_integer* expression containing an exponentiation operator if the exponent has a negative value.
- T7. Check that the value of `SYSTEM.MAX_INT` or `SYSTEM.MIN_INT` can be used successfully in *universal_integer* static expressions.
- T10. Check that static *universal_real* expressions are evaluated exactly. In particular, check that:
 - sums, differences, products, quotients, and exponentiations of small rational numbers are performed correctly.

Implementation Guideline: Use numbers such as 2.0/7.0, and check that equivalent values are considered equal, i.e., that if a rational arithmetic package is used, numbers are reduced correctly to lowest terms. In addition, check that expressions such as $0.1/0.2 = 0.5$ evaluate as true.

Implementation Guideline: Check that an integer literal can be multiplied by a real literal (two cases) and that a real literal can be divided by an integer literal.

- cascading use of fractional values does not lose precision, e.g., compute the terms in a series approximation; compute various constants based on PI.
- illegal constructs dependent on exact evaluation of a static *universal_real* expression are detected correctly.
- values of 'SMALL and 'LARGE can be computed correctly as static expressions.

T11. Check that nonstatic *universal_real* expressions are evaluated with the accuracy of the most precise predefined floating point type (i.e., the type for which DIGITS equals SYSTEM.MAX_DIGITS).

T12. Check that NUMERIC_ERROR (or CONSTRAINT_ERROR; see AI-00387) is raised for a *universal_real* expression if division by zero is attempted.

Implementation Guideline: At least one case should use all static operands.

Implementation Guideline: The divisor should have type *universal_integer* as well as type *universal_real*.

Check that CONSTRAINT_ERROR is raised for $0.0^{*(-1)}$ (or any other negative exponent value).

T13. Check that NUMERIC_ERROR (or CONSTRAINT_ERROR; see AI-00387) is raised for a nonstatic *universal_real* expression if the value would lie outside the range of the base type of the most accurate predefined floating point type and MACHINE_OVERFLOW is true for that type.

Implementation Guideline: Report whether or not the exception is raised even if MACHINE_OVERFLOW is not true.

Check that if MACHINE_OVERFLOW is true, NUMERIC_ERROR (or CONSTRAINT_ERROR; see AI-00387) is raised for a nonstatic *universal_real* expression if a static operand has a value outside the range of the most accurate floating point base type, even if the result lies within the range of the base type.

T14. Check that when rounding static *universal_real* values to an integer type, the rounding is performed correctly.

Implementation Guideline: Use both positive and negative values.

Check how rounding is performed when the *universal_real* value is halfway between adjacent integer values.

Implementation Guideline: Among the possible rounding modes are: round to +infinity, round to -infinity, round to zero, round away from zero, round to even, and round to odd.

T16. Check that for the relational and equality operators, one operand cannot have type *universal_integer* if the other has type *universal_real*.

Check that for membership tests of the form $E \text{ In } L..R$ and $E \text{ not in } L..R$, two of the expressions E, L, and R cannot have the type *universal_integer* if the third has type *universal_real*, and vice versa.

Check that for the binary + and - operators, one operand cannot have type *universal_integer* if the other has type *universal_real*.

Check that neither operand of the rem and mod operators can have type *universal_real*.

Check that if the first operand of the division operator has type *universal_integer*, the second operand cannot have type *universal_real*.

Check that the second operand of the exponentiation operator cannot have the type *universal_real*.

T17. Check that relational expressions and membership tests are evaluated exactly when all operands have type *universal_real* and at least one operand is static.

Chapter 5

Statements

5.1 Simple and Compound Statements

Semantic Ramifications

S1. A sequence of statements must contain at least one statement, even if it is only a null_statement.

S2. The occurrence of a label, e.g., <<I>>, should not be confused with the declaration of a label. <<I>> refers to the implicit declaration of I as a label; the context in which <<I>> occurs determines where this implicit declaration occurs. Consider the following example:

```
for I in 1..10
loop
<<I>>    null;          -- illegal
end loop;
```

The occurrence of label I is illegal because the only visible declaration of I is the loop parameter's declaration, and a loop parameter is not a label. Label I is implicitly declared in an enclosing declarative part. Normal visibility rules then apply to explicit occurrences of this implicitly declared identifier. In particular, <<I>> is only legal when the implicit declaration of I as a label is visible.

S3. The scope of a label, a block name, or a loop name begins in the declarative part of the body or the block where its implicit declaration appears. In particular, the scope begins before the occurrence of the name in the statements of the body or block:

```
declare

    LAB : INTEGER;          -- object named LAB

    procedure P is
        -- label LAB declared here
    begin
        LAB := 1;           -- illegal: LAB denotes the following label,
        <<LAB>> return;      -- not the preceding object
    end;

begin ... end;
```

The assignment to LAB is illegal because the object LAB is not directly visible anywhere within the statements of the body of P.

S4. Because label, block, and loop names are declared at the end of a declarative part, the identifier can be used earlier in the declarative part:

```
declare
    L : INTEGER := 0;
begin
    declare
        X : INTEGER := L;    -- legal
```

```

    begin
    <<L>>...
    end;
end;

```

There is no testable semantic effect associated with the requirement that such names be declared in the order of their occurrence in the text.

s5. The place where labels, block names, and loop names are declared also affects the use of these names in expanded names:

```

B0:  declare
      -- L1 and B2 are declared here
    begin
L1:  for I in 1..10
      loop
B2:  declare
      X : INTEGER;
      begin
        X := B0.L1.I;      -- legal
        X := B0.L1.B2.X;   -- illegal
        X := B0.B2.X;      -- legal
      end B2;
    end loop L1;
  end B0;

```

Both L1 and B2 are declared immediately within block B0. Since B2's declaration as a name does not occur within the loop, it is illegal to write L1.B2.X. Instead, B0.B2.X must be written. Note that since the loop parameter is declared immediately within the loop (RM 5.5/6), the notation L1.I is allowed.

s6. The requirement that labels, block names, and loop names be distinct within a program unit still allows such a name to be identical to the name of a variable, exception, etc. declared within the program unit:

```

procedure P is
  X : INTEGER;
begin
  begin
    -- block
    <<X>> ...      -- legal (1)
    <<Y>> ...
  end;
  begin
    <<Y>> ...      -- illegal (2)
  end;
end P;

```

The label at (1) is legal because its identifier is declared within the enclosing block rather than within the enclosing procedure. The label at (2) is illegal because even though it is declared within its enclosing block, RM 5.1/4 requires that label names be distinct within subprogram P.

Changes from July 1982

s7. There are no significant changes.

Changes from July 1980

S8. Statement labels and loop or block identifiers are now declared within the innermost enclosing block instead of within the innermost enclosing program unit.

S9. The rules for transfer of control are expanded to include the semantics of a terminate alternative and the effect of an abort statement.

Legality Rules

- L1. Within the `sequence_of_statements` and the (optional) exception handling part of a subprogram body, package body, task body, or generic unit (and excluding any nested subprograms, packages, tasks, or generic units), no identifiers used for statement labels, block identifiers, or loop identifiers are allowed to be the same (RM 5.1/4).

Test Objectives and Design Guidelines

- T1. Check that declarations cannot be interleaved with statements in a `sequence_of_statements`.

Implementation Guideline: Try them in a `subprogram_body`, `package_body`, `block`, `if_statement`, `case_statement`, and `loop_statement`. In some cases, all declarations should precede all statements.

- T2. Check that the statements in a `sequence_of_statements` are executed in succession.

Implementation Guideline: Avoid raising exceptions and avoid using `exit`, `goto`, and `return` statements.

Check that the existence of labels on some of the statements in the sequence have no effect.

Check that multiple labels are permitted on a statement.

Check that a statement label may precede a named block or loop.

Check that labels are permitted at the beginning of a `sequence_of_statements` in every context that permits a `sequence_of_statements`, namely, in `accept` statements, blocks, loops, `case` statement alternatives, exception handlers, `if` statement alternatives, package, task, and subprogram bodies, `select` statements, and `select` alternatives.

Implementation Guideline: Use unique labels throughout these tests. (Checks for nonunique labels are performed in IG 8.3.a/T1.)

- T3. Check that an empty `sequence_of_statements` is not allowed in an `accept` alternative, `accept` statement, `block` statement, `case` statement alternative, `conditional` entry call, `delay` alternative, `exception` handler, `if` statement, `loop` statement, `package` body, `selective` wait, `subprogram` body, `task` body, and `timed` entry call.

- T4. Check that labels, loop identifiers, and block identifiers are implicitly declared at the end of the declarative part of the innermost block statement, subprogram body, package body, generic package or subprogram body, or task body that enclose them, and therefore:

- cannot be the same as other identifiers declared in the same declarative part (see IG 8.3.a/T6);
- can be the same as identifiers declared in an enclosing program unit or block (see IG 8.3.a/T2, IG 8.3.a/T6);
- cannot be written in a `goto` statement or in `<<>>` brackets within a `for` loop if the label's identifier (or block or loop identifier) is the same as the loop parameter identifier (see IG 8.3.a/T5);
- the label, loop, or block identifier can be used earlier in the declarative part to refer to an entity declared in an enclosing unit or block;
- if a named block is nested in a named loop, the loop name cannot be used in the prefix of an expanded name denoting an entity declared in the block;

- hide other declarations given in outer blocks or bodies;

Implementation Guideline: Check the case where an identifier is used as a label, a block name, and a loop name after its first use as a variable.

- T5. Check that labels, loop identifiers, and block identifiers must be distinct within a subprogram body, package body, task body, generic package body, or generic subprogram body even if the same identifier occurs in different blocks (see IG 8.3.a/T1).

5.2 Assignment Statements

Semantic Ramifications

S1. Although the target variable and the source expression must have the same type (i.e., the same base type), the expression may yield any value of that type. The value is then checked to see if it satisfies any subtype constraint imposed on the target variable. If the check fails, an exception is raised. Such an exception must be raised before any portion of the target variable is updated.

S2. The evaluation of an assignment statement proceeds as follows:

1. In any order:
 - a. evaluate the source expression (and hold the entire value in a nonoverlapping temporary variable);
 - b. evaluate any expressions appearing in the target variable and check that the target variable names an existing object (e.g., check that subscript values are within the bounds of the array object being subscripted, etc.).
2. Check that the expression value satisfies any subtype constraints imposed on the target variable.
3. Update the target variable with the evaluated expression value. (This step is only performed if none of the preceding steps raises an exception.)

S3. When performing the checks in step 2, no exception can be raised by the actions taken to perform the check itself. For example, `NUMERIC_ERROR` must not be raised when checking assignments to `INTEGER` variables.

S4. The order of evaluating the identity of the target variable and the primaries of the expression is not defined by the language. This means that the identity of the target variable can be changed as a result of evaluating the expression:

```

A    : array (1..4) of INTEGER;
I    : INTEGER := 4;
function FI return INTEGER is
begin
    I := 5;
    return 3;
end FI;
...
A(I) := FI;

```

This program is not erroneous, although its effect depends on whether `FI` is evaluated before or after `A(I)`. If evaluated before, then `A(I)` will raise `CONSTRAINT_ERROR`. If `FI` is evaluated after `A(I)` (or after the value of `I` is determined), then `FI`'s value will be assigned to `A(4)`.

Regardless of the evaluation order, it is not acceptable to assign a value to a nonexistent component of A, namely A(5).

s5. Although dependence on the order of evaluation does not, in general, make a program erroneous, the RM explicitly specifies one circumstance under which evaluation order can make a program's execution erroneous; namely, a program is erroneous if the evaluation of an assignment expression changes the discriminant of a target variable:

```

type DISCRIM is (INT, FLT);
type VR (D : DISCRIM := INT) is
  record
    case D is
      when INT =>
        I : INTEGER;
      when FLT =>
        F : FLOAT;
    end case;
  end record;

R : VR;

function F return INTEGER is
begin
  R := (D => FLT, F => 2.0);
  return 1;
end F;

R := (D => INT, I => 0);
R.I := F;           -- erroneous

```

Since the evaluation of F changes the discriminant of R, the assignment is erroneous. Because the RM defines this situation as erroneous, an implementation is allowed to assume that the expression does not change the discriminant of the target variable. Consequently, an implementation can evaluate R.I (checking that I exists for the R's current discriminant value), can evaluate F, and can then assign F's value to the location normally occupied by R.I, even though this assignment may produce an invalid floating point value for R.F. Note that if F is evaluated first, then the evaluation of R.I will raise CONSTRAINT_ERROR, since I is not a valid selector when R.D = FLT (RM 4.1.3/8).

s6. Although the RM says that the value of the expression is checked against the subtype of the variable after the expression is evaluated, an optimizer can perform the subtype check earlier. RM 11.6/4 allows predefined operations in assignment statements to be evaluated as soon as possible, and the act of checking a discriminant constraint or array length involves the use of predefined equality operations. For example:

```

type ARR is array (INTEGER range <>) of INTEGER;

type REC (D : INTEGER) is
  record
    X : INTEGER;
  end record;

ARR_V : ARR (1..3);
REC_V : REC (3);
GLOBAL : INTEGER := 1;

```

```

function F return INTEGER is
begin
    GLOBAL := GLOBAL + 1;
end F;
...
ARR_V := (1..4 => F);
REC_V := (D => 4, X => F);

```

F need not be invoked in either of these assignments since in the first assignment, the length of the array value can be determined to exceed the length of ARR_V prior to evaluating F. Similarly, in the second assignment, the discriminant of the aggregate can be determined to be unequal to the discriminant of REC_V before F is invoked. In both cases, CONSTRAINT_ERROR can be raised before F is evaluated. Consequently, GLOBAL can have the value 1 when an exception handler is entered.

Changes from July 1982

S7. There are no significant changes.

Changes from July 1980

S8. It is now explicitly stated that the value of an object remains unchanged when the assignment operation raises CONSTRAINT_ERROR.

S9. Assignment to a subcomponent of an object of an unconstrained record type is erroneous if evaluation of the expression changes the value of the discriminant.

S10. For array variables, a subtype conversion is applied to the expression before checking that the subtype of the expression belongs to the subtype of the variable.

Legality Rules

- L1. The base types (not necessarily the subtypes) of the target variable and the source expression must be the same.
- L2. The object being assigned to must not have been declared as a constant in an object declaration, as an in formal parameter of a subprogram or generic unit, or as a subcomponent of such an object. It must not be a literal, an expression, a loop parameter, a discriminant component of a record or private type, nor can it be a function_call, or an attribute.
- L3. If the name being assigned to has the form of an identifier, the identifier cannot be the name of an enumeration literal, subprogram, entry, block, loop, statement label, package, pragma, task, type, or subtype.
- L4. The type of the variable must be a type for which assignment is declared, i.e., it cannot be a task type or a limited private type, nor can it be a composite type having a component of a type for which assignment is not declared.

Exception Conditions

Exceptions raised during the evaluation of the variable and expression are covered in Chapter 4.

- E1. CONSTRAINT_ERROR is raised if an assignment to a scalar variable would violate the variable's range constraint.
- E2. CONSTRAINT_ERROR is raised for the assignment of array types; see IG 5.2.1/E1.
- E3. CONSTRAINT_ERROR is raised if a value of a discriminant of the expression does not

equal a corresponding discriminant value of the variable, and if the variable has a constrained record or private type or is an object designated by an access value.

- E4. **CONSTRAINT_ERROR** is raised if the variable has a constrained access type, the value being assigned is not null, and:
- any index bound of the designated object does not equal the corresponding bound specified for the access type's constraint.
 - any discriminant of the designated object does not equal the corresponding value specified for the access type's constraint.

Test Objectives and Design Guidelines

- T1. Check that an **assignment_statement** replaces the current value of the target variable with the value of the source expression. Check this for **INTEGER**, **BOOLEAN**, **CHARACTER**, **FLOAT** (separately), a fixed type (separately), another enumeration type, **STRING**, another array type, a record type (with and without discriminants), and an access type.

Implementation Guideline: Include components with a constraint that depends on a discriminant.

- T2. Check that the left side (target) of an **assignment_statement** must be a variable.

Implementation Guideline: Try the following kinds of entities as the target: a declared constant (including components of a constant); a generic in parameter; a subprogram in parameter; a loop parameter; a named number; a function call returning a result of a scalar, array, record, access, and private type; a sliced function result; a component of a composite function result; and a record discriminant.

- T3. Check that multiple assignments are not permitted within a single **assignment_statement**; in particular, check that the following forms are prohibited:

```
variable, variable := expression;
variable := variable := expression;
variable := expression operator (variable := expression);
```

- T4. Check that the types of the target variable and source expression must match at compile time.

Implementation Guideline: Try combinations such as:

```
INTEGER vs. FLOAT
INTEGER vs. LONG_INTEGER
INTEGER vs. CHARACTER
array of INTEGER vs. array of FLOAT
arrays and slices
access record vs. record
```

- T5. Check that, for scalar types (**INTEGER**, **BOOLEAN**, **CHARACTER**, **FLOAT**, a fixed type, and a user-defined enumeration type), **CONSTRAINT_ERROR** is raised when the expression's value is outside the target variable's range; check that the value of the target variable is not altered.

Implementation Guideline: Try some subtests with ranges that can be checked at compile time and others with ranges that must be checked at run time.

Implementation Guideline: Use values just inside and just outside the target range (at both ends).

- T6. Check that the equality operator (**=**) cannot be used as an assignment symbol.
- T7. Check that **NUMERIC_ERROR**, **CONSTRAINT_ERROR**, or any other exceptions are not raised for **INTEGER** assignments when the expression's value is actually in the target variable's range, and when the target variable's range and/or the expression's value are near **INTEGER'FIRST** or **INTEGER'LAST**.
- T8. Check that a record variable constrained by a specified discriminant value cannot have its

discriminant value altered by an assignment. Assigning an entire record value with a different discriminant value should raise **CONSTRAINT_ERROR** and should leave the target variable unaltered.

Implementation Guideline: Try both static and nonstatic discriminant values.

- T9. Check that a record variable designated by an access value cannot have its discriminant altered, even by a complete record assignment, and even though the target access variable is not constrained to designate an object with a specific discriminant value. In other words, check that the attempt to change the target's discriminant raises **CONSTRAINT_ERROR** and leaves the target record unaltered.

Implementation Guideline: Try both static and nonstatic discriminant values.

Implementation Guideline: The discriminants of the designated type should have default values in the nonstatic case.

- T10. Check that record assignments use "copy semantics," i.e., that values of the target variable can be used in the expression being assigned (see IG 5.2.1/T2 for the equivalent array variable test).

Implementation Guideline: In particular, try the following:

```
type REC is record
  X, Y : INTEGER;
end record;

R := (0, 0);
R := (X => 1, Y => R.X);  -- (1, 0)
...
R := (X => R.Y, Y => 2);  -- (0, 2)
```

- T11. Check that index and discriminant constraints for assignment of access subtypes are checked, as in:

```
type T is access an_unconstrained_type;
subtype S1 is T constraint1;
subtype S2 is T constraint2;
W      : T;
X1, X2 : S1;
Y1, Y2 : S2;
```

Check that:

- any of the above variables can be assigned to each other if the value being assigned is null.
- X1 or X2 can be assigned to each other or to W.
- **CONSTRAINT_ERROR** is raised if X1 is assigned to Y1 and X1 is not null.
- **CONSTRAINT_ERROR** is raised if W is assigned to X1, W is not null, and the constraints of the object designated by W do not equal the constraints imposed on S1.
- null can be assigned to any of these variables.

- T12. Check that if the evaluation of the expression in an assignment statement raises an exception, the value of the target variable is not changed.

Implementation Guideline: Use a case where a record or array aggregate raises an exception after the first component is evaluated, when the first component is a literal.

Implementation Guideline: Also try catenation:

```
S := T (2 .. 5) & T (7 .. I + J);
```

where $I + J$ raises **NUMERIC_ERROR** or the slice raises **CONSTRAINT_ERROR**.

- T13. Check whether the entire expression is evaluated before checking that a variable's discriminant constraint is satisfied, or that an array value has the correct number of components for each dimension.

5.2.1 Array Assignments

Semantic Ramifications

- S1. The base type of a slice is the base type of the array denoted by the prefix, but the subtype (i.e., the index constraint) is determined by the upper and lower bounds of the slice.
- S2. When checking that the lengths of corresponding dimensions match, if the lengths do match but happen to exceed `INTEGER'LAST` or `SYSTEM.MAX_INT`, then the length checking must succeed and must not raise `NUMERIC_ERROR` or `CONSTRAINT_ERROR`. For example, if an implementation does not raise `STORAGE_ERROR` or `NUMERIC_ERROR` for the following declarations:

```
type A is array INTEGER range <> of BOOLEAN;
pragma PACKED (A);
X : A (-1..INTEGER'LAST);
Y : A (-2..INTEGER'LAST-1);
```

Then it must not raise `NUMERIC_ERROR` when executing:

```
X := Y;
```

or

```
X := (-2..INTEGER'LAST-1 => TRUE);
```

- S3. The semantics of assigning overlapping slices is equivalent to first assigning the expression value to a nonoverlapping temporary variable and then assigning the temporary to the target.

S4. Only the index constraints may differ for different objects of a given array type since only index constraints can be applied to an array type. Thus, since the component type constraints must be the same for all objects of the array type, it is not necessary to check the component constraints in an array assignment. Any component constraint checking of the source expression value will already have been performed as part of expression evaluation semantics, which must yield a value of the target array type. In particular, if the source expression is an array aggregate, then the individual component values are constraint checked as the (temporary) array value is formed (see RM 4.3.2/11), after which the array value is assigned to the array target variable.

S5. An implementation may directly assign the components of an array aggregate to the components of an array target (without constructing a temporary array value) if all of the component values are constraint checked (and exceptions raised) before any component of the target variable is updated (see IG 5.2/S).

- S6. If given these declarations,

```
type DAY is (MON, TUE, ..., SUN);
type ARR is array (MON..FRI) of INTEGER;
type ARR_DAY is array (DAY range <>) of INTEGER;
NORM      : ARR;
NORM_DAY : ARR_DAY (MON..FRI);
```

then `NORM_DAY := (WED..FRI => 0, SAT..SUN => 1)` does not raise `CONSTRAINT_ERROR`; the aggregate has the correct length and the implied subtype conversion provides the bounds `MON..FRI`. (Note that after the assignment, `NORM_DAY(FRI) = 1` because of the subtype conversion.) On the other hand,

```
NORM := (WED .. SUN => 0);
```

raises `CONSTRAINT_ERROR` since the bounds of the aggregate, `WED..SUN`, do not all belong to the index subtype of `ARR`. (The index subtype is `MON..FRI`, not `DAY`, (see RM 3.6/7) and the bounds of the aggregate are checked against this subtype prior to the assignment statement's subtype conversion (RM 4.3.2/11)).

S7. The required subtype conversion checks that if the subtype of the variable has a null range for at least one dimension, the expression must denote a null array value (and vice versa), but corresponding dimensions need not have the same index values. The RM only requires that there be matching components. Since a null array has no component for any dimension, the rule does not imply corresponding dimensions must have the same bounds.

Changes from July 1982

S8. There are no significant changes.

Changes from July 1980

S9. The value of the expression is implicitly converted to the subtype of the array variable.

Exception Conditions

- E1. `CONSTRAINT_ERROR` is raised for the assignment of a non-null array value to a non-null array variable if the number of components for corresponding dimensions is not equal.
- E2. `CONSTRAINT_ERROR` is raised if a non-null array value is assigned to an array variable having a null index range for one of its dimensions, or if a null array value is assigned to a non-null array variable.

Test Objectives and Design Guidelines

- T1. Check that array subtype conversion is applied after an array value is determined.
Implementation Guideline: Use the `NORM := (WED..SUN => 0)` example.
- T2. Check that the assignment of overlapping source and target variables (including arrays and slices in various combinations) satisfies the semantics of "copy" assignment, i.e., it is equivalent to first copying to a nonoverlapping temporary variable.
Implementation Guideline: Try both static and dynamic bounds so as to allow and prevent compile-time detection of the overlap. Also try the following "overlapping" aggregate assignment:

```
A : array (1..4) of INTEGER := (1..4 => 0);
A := (1, A(1), A(1), A(1));
```

Also try the following "overlapping" catenation of slices:

```
S : STRING (1..10) := "ABCDEFGHIJ";
S := 'K' & S(1..2) & S(1..2) & S(1..5);
```

Also try the above for an array of integers. Each kind of overlap should be tested in both directions, i.e., target bounds less than source bounds and target bounds greater than source bounds.

- T3. Check that array assignments with matching lengths do not raise `CONSTRAINT_ERROR` and are performed correctly. Try all meaningful combinations of the following:
 - a. Static vs. nonstatic.

1. Static -- Use static bounds and indices so that the length check can be done at compile time.
2. Nonstatic -- Use nonstatic bounds and indices whose values can only be known at run time so as to force a run-time length check.

b. Arrays and slices.

1. Array -- Use an entire array, i.e., don't subscript or slice it. Try both non-null and null arrays. For the source expression, try both array objects and array aggregate.
2. Slice -- Try both non-null and null slices.

Also try to mix these within single assignment statements (as to kind of target and/or source) whenever the types match. For example:

```
S1 : STRING(1..10);
S2 : STRING(1..20);
...
S1 := S2(6..15);
```

Implementation Guideline: Check that all components are correctly assigned to the proper positions. For lengths that exceed INTEGER'LAST, check that the length checking does not raise an exception, such as NUMERIC_ERROR or CONSTRAINT_ERROR.

- T4. Check that array assignments with nonmatching lengths raise CONSTRAINT_ERROR. Try all meaningful combinations of the following:

a. Static vs. nonstatic.

1. Static -- Use static bounds and indices so that the length check can be done at compile time.
2. Nonstatic -- Use nonstatic bounds and indices whose values can only be known at run time so as to force a runtime length check.

b. Arrays and slices.

1. Array -- Use an entire array, i.e., don't subscript or slice it. Try both non-null and null arrays. For the source expression, try both array objects and array aggregate.
2. Slice -- Try both non-null and null slices.

Also try to mix these within single assignment statements (as to kind of target and/or source) whenever the types match. For example:

```
S1 : STRING(1..10);
S2 : STRING(1..20);
...
S1 := S2(5..15);      -- CONSTRAINT_ERROR
```

Implementation Guideline: Check that when an exception (CONSTRAINT_ERROR) is raised, the target variable is not altered.

5.3 If Statements

Semantic Ramifications

S1. Because RM 5.3/3 uses lower case in specifying the type of condition expressions, conditions may have a type derived from STANDARD.BOOLEAN as well as the predefined BOOLEAN type (see RM 3.5.3/1). Since conditions occur in loop statements, exit statements, and select statements, expressions of a derived boolean type are allowed in these contexts as well as in if statements.

Changes from July 1982

S2. There are no significant changes.

Changes from July 1980

S3. Condition expressions can have any boolean type, including types derived from BOOLEAN.

Test Objectives and Design Guidelines

Short-circuit evaluation of conditions is tested in IG 4.5.1.c/T22.

T1. Check that every if_statement must end with end if.

T2. Check that else cannot precede elsif in an if_statement.

T3. Check that elsif cannot be spelled as elseif, elif, elif, or else_if.

T4. Check that condition expressions must have a boolean type.

Implementation Guideline: In particular, ensure that the integer literal zero (0) is illegal and that a type derived from BOOLEAN is legal.

T5. Check that control flows correctly in basic non-nested if_statements that have no elsif or else parts.

Implementation Guideline: Since this form of if_statement is heavily used in the other executable tests to conditionally invoke the routine FAILED, it must be carefully tested here. Thus, the order of the following subtests is critical.

First do all of the following subtests where the condition has the value false. Each subtest has the form:

```
if false_condition then
  FAILED ("message");
end if;
I := I + 1;
```

This checks that the then action is correctly skipped. The variable I should be initialized to zero before the first subtest. Its purpose here is to verify that control skips to whatever follows the then action.

Use the following increasingly complex forms for the conditions: a boolean literal (e.g., FALSE), a named boolean constant or variable (e.g., B), a simple relation (e.g., A > 0), a logical expression (e.g., B and A > 0), and a condition with and then or or else. Do not use conditions that depend on short-circuit evaluation.

All of the above subtests must be done twice: first with static conditions, and second with nonstatic conditions.

Next, all of the subtests (for all the above forms of conditions and for both static and nonstatic conditions) must be repeated, but with the conditions having the value true. Each of these subtests has the form:

```
if true_condition then
  I := I + 1;
  goto Ln;
end if;
FAILED ("message");
<<Ln>> null;
```

This checks that the then action is correctly initiated.

Next, do the following subtest:

```

if static_true_condition then
    I := I + 1;
end if;

if dynamic_true_condition then
    I := I + 1;
end if;

if I /= n then
    FAILED ("message");
end if;

```

where *n* is an integer literal whose value is the number of *false_condition* subtests plus the number of *true_condition* subtests and *I* is initially 0. This, combined with the previous *false_condition* and *true_condition* subtests, checks that the *then* action is either skipped correctly or is initiated and completed correctly (for non-null actions).

Last, try four subtests where the *sequence_of_statements* is the *null_statement*, namely, with a *static_true_condition*, with a *dynamic_true_condition*, with a *static_false_condition*, and with a *dynamic_false_condition*.

- T6. Check that control flows to the correct alternative *sequence_of_statements* in complex non-nested *if_statements*. Try at least the following forms:

```

a.  if ...
    else ...
    end if;

b.  if ...
    elsif ...
    elsif ...
    end if;

c.  if ...
    elsif ...
    elsif ...
    else ...
    end if;

```

Implementation Guideline: Each of the above forms must be tried repeatedly for each possible control flow choice. Thus (b) above needs four subtests, one for each of the four choices: (1) *if* selected, (2) first *elsif* selected, (3) second *elsif* selected, and (4) none selected. Occasionally, an alternative *sequence_of_statements*, whether selected or not, should consist of a *null_statement*.

For the conditional expressions (conditions), start out with simple boolean variables (e.g., *B*) or constants (e.g., *TRUE*). Then try simple relational expressions (relations, e.g., *A > 0*). Then, try logical expressions (e.g., *A > 0* and *B*). Last, try conditions involving *and* then *and* or *else*, but don't use conditions that depend on short circuit evaluation. However, it is not necessary to try each of these condition forms with each of the *if_statement* forms.

All of the above subtests must be done twice: once with static conditions and once with dynamic conditions.

It can be assumed that the basic non-nested *if_statement* with no *elsif* or *else* parts, and with the condition forms described above, works correctly (see IG 5.3/T5). Thus, this basic form can be used to verify whether the above forms work, and to invoke the routine *FAILED* when they have failed.

- T7. Check that control flows correctly in simple nested *if_statements*.

Implementation Guideline: Try about four subtests involving doubly or triply nested *if_statements*. Use mixtures of the forms of *if_statements* and of static and dynamic conditions mentioned in IG 5.3/T5 and IG 5.3/T6. Also try some null alternatives, i.e., where the *sequence_of_statements* is a single *null_statement*, sometimes selected and sometimes not. It can be assumed that non-nested *if_statements* work correctly (see IG 5.3/T5 and IG 5.3/T6).

T8. Check that control flows correctly in complex nested if_statements.

Implementation Guideline: This is intended to be a worst-case test in terms of probable actual user programs. Try a nesting of if_statements that is at least ten deep along the selected path and at least five deep along a few of the nonselected paths. Along the selected path, use a mixture of all of the forms of if_statements and static and dynamic conditions mentioned in IG 5.3/T5 and IG 5.3/T6. Likewise, use mixtures of these forms in the nonselected paths. The alternative sequences_of_statements should include the following forms:

- a. a null_statement only,
- b. an assignment_statement only,
- c. an if_statement only,
- d. at least one assignment_statement and at least one if_statement in various orders.

It can be assumed that non-nested if_statements work correctly (see IG 5.3/T5 and IG 5.3/T6).

T9. Check that neither an if_statement nor any of its alternative sequences_of_statements implicitly introduces a new declarative region. In particular, check that an alternative cannot begin with a declarative_part; that names declared just outside an if_statement cannot also be declared as labels inside the if_statement; and that labels declared in one alternative of an if_statement cannot be declared as labels in another alternative of the if_statement (see also IG 8.3.a).

5.4 Case Statements

Basic properties of case statements are treated in IG 5.4.a. Interactions between the subtype of a case expression and the set of values covered by the alternatives are checked in IG 5.4.b.

Note that the vertical bar in the syntax production is a terminal symbol, not a meta-syntactic symbol (RM 1.5/(e)).

5.4.a Basic Case Statement Properties

Semantic Ramifications

S1. The scope of statement labels inside case statements is discussed in IG 8. Restrictions on the use of goto statements are discussed in IG 5.9/S and IG 8.

S2. Case expressions must have a discrete type, i.e., an integer, a predefined-enumeration type, a user-defined enumeration type, or a user-defined type derived from a discrete type. In particular, a private type whose corresponding full type declaration is a discrete type cannot be used in case expressions outside the scope of the declaration implementing the type. (Since case expressions must have a discrete type, strings (even strings of length one) and fixed point types with integral DELTAs, are not permitted.)

S3. Choices of the form

```
ST range L .. R
ST
A' RANGE (n)
```

are permitted in case statements (RM 3.7.3/2 and RM 3.6/2) (when ST denotes a discrete subtype).

S4. Choices must be simple expressions; hence, relational and logical operators cannot be used in choices unless the expression containing these operators is parenthesized.

S5. If a programmer wishes to obtain the effect of executing several alternatives for a single case expression value, he will have to write a statement of the form:

```

case I is
  when 1 => goto    FIRST;
  when 2 => goto    SECOND;
  when 3 => goto    THIRD;
  when others => goto REST;
end case;
<<FIRST>>      ... ; goto REST;
<<SECOND>>     ... ; -- fall through to THIRD
<<THIRD>>      ... ; goto DONE;
<<REST>>       ... ;
<<DONE>>       ...

```

An implementation may wish to provide special optimizations for such a program structure.

S6. An implementation may wish to use a jump-table implementation for alternatives whose non-others choices cover a small range of values and an if-then-else implementation when a large range of values is covered but only a few alternatives are present.

S7. There is no rule prohibiting vacuous choices, e.g., a choice denoting a null range of values is allowed and so is an others choice that can never be selected.

S8. When a case expression has a static subtype, values outside the range of the subtype are not allowed as non-null choices (RM 5.4/4). For example, given the usual definition of the type DAY, consider the derived type:

```
type WEEKDAY is new DAY range MON .. FRI;
```

The following case statement is illegal:

```

X: WEEKDAY;
...
case X is
  when SAT | SUN => ...    -- illegal
  when MON..WED => ...
  when THU..FRI => ...
end case;

```

S9. Since the subtype of a generic in out parameter is always its base type, regardless of the type mark used in the parameter's declaration (RM 12.1.1/1), all values of the parameter's base type are allowed as case choices, e.g.,

```

subtype FIVE is INTEGER range 1 .. 5;

generic
  X : in out FIVE;
procedure P;

procedure P is
begin
  case X is
    when 10 => ...          -- legal
    when others => ...
  end case;
end P;

```

If X were not a generic parameter or if X were a generic in parameter, then the choice, 10, would be illegal.

S10. The predefined equality operation determines which case alternative is selected:

```

package P is
  type T is limited private;
  function "=" (X, Y : T) return BOOLEAN;
private
  type T is new INTEGER;
end P;

package body P is
  X : T := 1;
  function "=" (X, Y : T) return BOOLEAN is
  begin
    return (INTEGER (X) - 1) = INTEGER (Y);
  end;
begin
  case X is
    when 1 => PUT ("OK");
    when 0 => PUT ("uses redefined =");
    when others => null;
  end case;
end P;

```

Note that in this example, the predefined equality operator for T is hidden but is nonetheless used in the case statement.

Changes from July 1982

S11. There are no significant changes.

Changes from July 1980

S12. Case statements with no alternatives are illegal.

S13. The type of the case expression must not be a generic formal type.

S14. When the case expression is the name of an object whose subtype is static, or is a qualified expression or type conversion whose type mark denotes a static subtype, no value outside the range of the subtype is allowed as a choice.

Legality Rules

- L1. The base type of the case expression and each choice must be the same.
- L2. The base type of the case expression must have a discrete type.
- L3. Every choice must contain only static expressions, i.e., for choices of the form E, L .. R, ST range L .. R, and ST, L, R, and E must be static expressions, and ST must be a static subtype (i.e., ST'FIRST and ST'LAST must be static).
- L4. Two choices must not have a value in common.
- L5. When the case expression is the name of an object whose subtype is static, or is a qualified expression or type conversion whose type mark denotes a static subtype, no value outside the range of the subtype is allowed as a choice.
- L6. An others choice, if present, must be the only choice given in the last alternative specified for a case statement.

- L7. The type of the case expression must be determinable independently of the values or types of the choices used in the case statement.
- L8. The type of a case expression must not be a generic formal type or a type derived (directly or indirectly) from a generic formal type.
- L9. Case statements must have at least one alternative.

Test Objectives and Design Guidelines

- T1. Check that (cf. IG 3.7.3/T1):
 - the reserved word **is** is required;
 - **when** cannot be replaced by **if**,
 | cannot be replaced by **or**,
 => cannot be replaced by **then**,
 end case cannot be replaced by **end**, **endcase**, or **esac**,
 is cannot be replaced by **of**;
 - when the case expression is a simple variable, the name of the variable cannot follow **end case**;
 - when a case statement is labeled, the label name cannot follow **end case**;
 - a case statement cannot be labeled like a loop or block, i.e., with an identifier followed by a colon;
 - the **others** choice must be the only choice given in the last alternative;
 Implementation Guideline: Try an **others** choice as the first and middle alternative, and try it as the first, middle, and last choice in a set of choices for the last alternative.
 - at least one alternative is required.
- T2. Check that an alternative must have at least one statement (see IG 5.1/T3), and that it can consist of one or more statements (implicitly checked by other tests in this section).
- T3. Check that the following types are permitted as the type of a case expression (see also T4 and T5) (cf. IG 3.7.1/T3):
 - **BOOLEAN**
 - **CHARACTER**
 - user-defined enumeration type
 - all predefined integer types
 - user-defined types derived from these types
 Implementation Guideline: Use an integer type, an enumeration type, and a derived discrete type in forming the derived types for this test.
- T4. Check that a limited or nonlimited private type whose full type declaration declares a discrete type can be used in a case expression within the scope of the full declaration (i.e., within the package that defines the type) (cf. IG 7.4/T4).
- T5. Check that nondiscrete types are unacceptable as case expressions. In particular, check strings of length 1, fixed point types with integral DELTAs, and private types implemented as discrete types but used outside the scope of the declaration defining the private type's implementation.
- T6. Check that static expressions (other than a numeric literal, a character literal, an enumeration literal, or a constant name) are allowed as case expressions (see also T21).

- T7. Check that a variable used as a case expression is not considered local to the case statement. In particular, check that the variable can be assigned a new value, and the assignment takes effect immediately (i.e., the case statement does not use a copy of the case expression).

Implementation Guideline: Use a discriminant of a variant record as the case expression and change the discriminant by assigning a new record to the variable. Then check that the discriminant field has the correct value (e.g., by attempting to access components unique to the new variant part).

- T8. Check that the type of a case expression cannot be a generic formal discrete or integer type.

- T9. Check that a choice may not be a component simple name.

- T10. Check that the base type of the case expression and the choice must not be different.

Implementation Guideline: Try a case expression that has type *universal_integer* and choices that have type *INTEGER*; also try expressions and choices where the expression is a derived type and the choice is the parent type (and vice versa).

Implementation Guideline: Include a case when the subtypes are different and the case expression's subtype is nonstatic.

- T11. Check that the predefined equality operation determines which case alternative is selected.

- T12. Check that when the case expression is the name of a constant or variable having a static subtype, or is a qualified expression or type conversion with a static subtype, no choice may have a value outside the subtype's range.

- T13. Check that if a case expression is a variable (including a generic in out parameter), a constant (including a generic in parameter), a qualified expression, a type conversion, a function invocation, an attribute, a static expression, or a parenthesized expression having one of these forms, and if the subtype of the variable, the constant, the qualified expression, or the type conversion is nonstatic, then any value of the expression's base type may appear as a choice.

Implementation Guideline: In particular, try function calls, attributes, and expressions that have a static subtype, and expressions whose values are outside the range of this subtype.

Checks involving discrete ranges

- T20. Check that every pair of choices must cover a disjoint set of values (cf. IG 3.7.3/T2).

Implementation Guideline: Use both single values and ranges of values, and check for overlapping values within a single alternative and between alternatives. Use overlapping ranges whose end points are different, e.g., 3..5 and 4..6 as well as ranges in which overlap occurs only at the end points. Use some examples in which a large range of values has to be checked for potential overlap. The choices should not all occur in monotonically increasing or decreasing order.

- T21. Check that nonstatic choice values are forbidden (cf. IG 3.7.3/T3).

Implementation Guideline: Try a variable whose range is restricted to a single value. Try a discrete_range of the form ST, where ST is a subtype name having at least one nonstatic bound, as well as choices of the form ST range L .. R and L .. R, where either L or R is nonstatic. Include a case where an exception is raised, e.g., *POSITIVE* range 0 .. 3. Include *A RANGE*.

- T22. Check that all legal forms of choice are permitted in case statements, and in particular, that forms like ST range L .. R, and ST are permitted (cf. IG 3.7.3/T4).

Implementation Guideline: Use the same subtype name in more than one choice.

- T23. Check that choices using named numbers and static constants are permitted (cf. IG 3.7.3/T4).

- T24. Check that choices denoting a null range of values are permitted.

Implementation Guideline: The vacuous alternatives should have null as its sequence of statements in one test and a non-null sequence of statements in a separate test.

Check also that an **others** alternative can be provided even if all values of the case expression have been covered by preceding alternatives (see also IG 5.4.b) (cf. IG 3.7.3/T5).

- T25. Check that out-of-range derived type values are not permitted as choices.
- T26. Check that choices within and between alternatives can appear in nonmonotonic order (cf. IG 3.7.3/T6).
- T27. Check that relational and logical operators are allowed as choices only if the expressions containing these operators are enclosed in parentheses (cf. IG 3.7.3/T7). Similarly, check the membership operations.
- T28. Check that non-null choices of the form `ST range L .. R` are illegal if either `L` or `R` is outside `ST`'s bounds and `ST`, `L`, and `R` are all static (cf. IG 3.7.3/T3) (see also IG 5.4.a/T21).

Optimization checks

- T41. Check that the flow of control in a case statement is appropriate.

Implementation Guideline: Use a case statement with a small enumeration type inside a loop and check that each alternative is executed in the appropriate sequence. Also check a case statement whose alternatives contain only goto statements.

- T42. Check that a case statement may have:

- a. a large number of potential choices grouped into a small number of alternatives, e.g.,

```
INTEGER'FIRST .. -101
-100
100
101..INTEGER'LAST
others
```

- b. a small range of choices grouped as a small number of alternatives (suggesting a jump-table implementation), e.g., alternatives using values in the range `1 .. 10`
- c. a sparse set of alternatives in a large range, e.g., `1, 2, 1_000, 5_000, 10_000 .. 10_200`.
- d. a few alternatives covering a large range, e.g., `1 .. 10_000, -9_000..0, 11_000..INTEGER'LAST`.
- e. a small range very far from 0, e.g., `10_001, 10_002, 10_003, 10_004`; (this permits a biased jump-table implementation)
- f. an **others** alternative that covers a small range of noncontiguous values, e.g., `MON` and `FRI`;
- g. an **others** alternative that covers several noncontiguous ranges of values, e.g., `1 .. 10, 15, 1000, -500 .. -400`, etc.

and that the appropriate alternatives are executed.

Overloading Checks

- T60. Check that if the case expression is an overloaded literal or function call, the case statement is considered illegal.

Implementation Guideline: Try a function overloaded with an `INTEGER` and `FLOAT` return value, a function returning a value of an `INTEGER` and a derived integer type, and an integer literal in a context containing a derived integer type.

T61. Check that when the case expression has the type *universal_integer*, it is not automatically converted to the integer type of the choices (see T10).

5.4.b When others Can Be Omitted

Semantic Ramifications

S1. An **others** alternative can always be omitted if every value in the base type of the expression is covered by the **non-others** choices. In certain cases, full coverage of the base type is neither required nor allowed. These cases arise when the case expression is:

- a name of a variable or constant whose subtype is static;
- a qualified expression or a type conversion whose type mark denotes a static subtype.

In particular, note that the following forms of expression require coverage of the base type:

- a parenthesized expression;
- a function call, even if the return type is a static subtype; and
- an attribute of a static subtype.

Hence, if we have:

```
I: INTEGER range 1 .. 5;
subtype FIVE is INTEGER range 1 .. 5;
J: FIVE;
```

then any of the following case statements must only have choices covering the range 1 through 5:

```
case I is
case J is
case FIVE'(I+1) is
```

Note, in particular, that the expression $I + 0$ is not a name, a qualified expression, or a type conversion; hence, **case** $I+0$ **is** must provide choices for all values in the range of INTEGERS. However, since FIVE is a static subtype, the case expression $FIVE'(I+0)$ requires covering only values in the range 1 through 5.

S2. There is an important interaction between the use of a loop parameter as a case expression and the need for an **others** alternative, e.g., consider:

```
for I in FIVE range 1 .. 4 loop
  case I is
```

the subtype of I is static; hence, only the values 1 .. 4 need be covered.

S3. The subtype of a name is never contextually modified. In the inner case statement of the following example:

```
for I in 1 .. 100 loop
  case I is
    when 1 .. 10 =>
      case I is
        -- must still cover 1 .. 100
```

the inner case statement must still cover the range of I's static subtype (i.e., 1 .. 100) before others can be omitted.

S4. When an implementation discovers that an **others** alternative has incorrectly been omitted, it would be helpful to users if the values of the missing alternatives were indicated in a diagnostic message. This, however, is not a language requirement.

S5. Note that if A is an array type or object, A'FIRST and A'LAST are names whose subtype is defined by A's index subtype:

```
type TEN is range 1 .. 10;
type ARR is array (TEN range <>) of INTEGER;
```

```
A : ARR (1 .. 5);
B : ARR (1 .. N);           -- N nonstatic
```

A's index subtype is TEN. Since TEN is a static subtype, A's index subtype is static. However, if the case expression is the name A'FIRST, the choices must cover the range of the index base type, i.e., TEN'BASE, since though A'FIRST is a name, it is not the name of an object, and thus its subtype does not determine the range or permitted choices. In particular, it is not sufficient to cover just the range of TEN'FIRST .. TEN'LAST, i.e., 1 .. 10.

Changes from July 1982

S6. There are no significant changes.

Changes from July 1980

S7. When the case expression is a function with a static return type or a parenthesized expression with a static subtype, the set of case choices must cover the range of the base type.

Legality Rules

L1. The **others** alternative must be present if the set of values covered by the set of non-others choices does not equal:

- the set of values associated with the base type of the expression if either of the following two cases does not apply;
- the set of values associated with the subtype of a variable or constant whose name is given as the case expression, when this subtype is static;
- the set of values associated with a static subtype whose type mark is used in a qualified expression or type conversion serving as the case expression.

Test Objectives and Design Guidelines

T1. Check that if a case expression is a constant (including a generic in parameter having a nongeneric type), a variable, a type conversion, or a qualified expression, and the subtype of the expression is static, an **others** choice can be omitted if all values in the subtype's range are covered, and must not be omitted if one or more of these values are missing.

Implementation Guideline: The interaction between loops and case statements is tested separately below and in IG 5.5.b/T11-T14.

Implementation Guideline: One form of variable should be a selected component of a record and an object designated by an access value.

T2. Check that if a case expression is a variable (including a generic in out parameter), a constant (including a generic in parameter), a type conversion, a qualified expression, a function invocation, an attribute (in particular, 'FIRST and 'LAST), or a parenthesized expression having one of these forms, and the subtype of the expression is nonstatic,

others can be omitted if all values in the *base type's* range are covered, and must not be omitted if one or more of these values are missing.

Implementation Guideline: Do not use loop parameters as case expressions in this test; these are tested in IG 5.5.

- T3. Check that when the case expression is a loop parameter, an **others** alternative can be omitted under the appropriate circumstances (see IG 5.5.b/T12, /T13).
- T4. Check that even when the context indicates that a case expression covers a smaller range of values than permitted by its subtype, an **others** alternative is required if the subtype value range is not fully covered.
Implementation Guideline: Use the nested case statement example above.
- T5. Check that if the case expression is $I+0$, the full range of INTEGER values must be covered if I is an INTEGER type or an integer subtype.
- T6. Check that if the case expression is an enumeration literal, all the values of the literal's base type must be covered if **others** is omitted.

Check that if the case expression is an integer literal, **others** cannot be omitted, even if the alternatives cover SYSTEM.MIN_INT .. SYSTEM.MAX_INT.

5.5 Loop Statements

The discussion of loops is divided into four subsections:

- a. properties of all loops
- b. FOR loops
- c. WHILE loops
- d. continuous loops

5.5.a Properties of All Loops

Semantic Ramifications

Changes from July 1982

- S1. There are no significant changes.

Changes from July 1980

- S2. There are no significant changes.
- S3. The rule specifying where loop identifiers are implicitly declared is given in RM 5.1/3.

Legality Rules

- L1. If a loop is named, an identifier must be present after **end loop** and must be the same as the identifier naming the loop (RM 5.5/3).
- L2. Within the `sequence_of_statements` and the (optional) exception handling part of a subprogram body, package body, task body, or generic unit (and excluding any nested subprograms, packages, tasks, or generic units), no identifiers used for statement labels, block identifiers, or loop identifiers are allowed to be the same (RM 5.1/4).

Test Objectives and Design Guidelines

T1. Check the basic syntactic requirements:

- a loop can have a loop identifier, and if present, the same identifier must be present at the end of the loop; if not present, no identifier is permitted at the end of the loop;
- the identifier at the end of the loop cannot be a name of the form P.LABEL, where LABEL is the name of the loop;
- the identifier at the end of the loop cannot be the name of the loop_parameter or the name of a statement label preceding the loop statement;

Implementation Guideline: For the statement label case try:

```
<<A>> loop ... end loop A;      -- illegal.
```

- when a loop with a loop identifier is nested inside another loop with a loop identifier, the inner loop cannot be terminated by an end loop that mentions just the outer loop's identifier;

- the forms

```
loop for I in 1..10; ... end loop;
loop while X; ... end loop;
```

are prohibited;

- the reserved word loop cannot be replaced by do or a semicolon as in:

```
for I in 1 .. 10 do .. end [for];
while A do ... end [while];
for I in 1 .. 10; ... end;
while A; ... end;
```

- a loop cannot be terminated just with end (i.e., end loop is indeed required);
- check that end for and end while are illegal;
- check that the forms:

```
for I in 1..10 while A loop ... end loop;
while A for I in 1..10 loop ... end loop;
```

are not permitted;

- check that repeat until statements are illegal;
- check that ':=' cannot be used instead of in;
- check for illegal use of '=' in loop statements (PL/I style loop);
- check for illegal use of to in loop statements;
- check for illegal use of from and to in loop statements;
- check that loop cannot be followed by ';';
- check that a loop body cannot be empty (see IG 5.1/T3);
- check that a loop body can consist of more than one statement (Note: This check is performed implicitly as a result of coding other tests required for loops);

- check that loop parameters can only be simple names;
- check that **end loop** cannot be replaced by **endloop** or **pool**.

T3. Check that several levels of loop nesting are permitted. Design a capacity test to ensure that an arbitrary degree of loop nesting is permitted.

5.5.b FOR Loops

Semantic Ramifications

S1. If a `loop_parameter` is renamed inside a block in a loop, the new name shares the properties of the `loop_parameter`; namely, the new name cannot be used in an assignment context. See IG 8.5/S for further discussion.

S2. The `loop_parameter_specification` declares the loop parameter as an object whose base type and subtype are defined by the discrete range. The subtype of a `loop_parameter` is only of interest when the `loop_parameter` is used in a case statement, because the subtype of the case expression determines when an **others** alternative can be legally omitted and the permitted range of choice values (see IG 5.4/S).

S3. Consider the following situation:

```
for I in L .. R
  case I is
```

What is the range of values that must be covered by the case statement alternatives? The range is determined by expanding the loop statement into the form:

```
for I in T range L .. R
  case I is
```

and then treating `I` in the case statement as though it had been declared (AI-00006):

```
I : T range L .. R;
```

`T` is determined by the base type of `L` and `R`, without regard to the subtype of `L` and `R` (see RM 3.6.1/2). Moreover, if `L` and `R` are both composed solely of integer literals, then `T` is the predefined type `INTEGER` (RM 3.6.1/2).

S4. Given that the subtype of `I` is determined, then the legality of the case statement follows the rules discussed in IG 5.4/S. In particular, the legality of case statements having no **others** alternative or having choices outside the range `L .. R` is affected by whether `I` has a static subtype.

S5. An implementation must be careful to avoid raising `NUMERIC_ERROR` or `CONSTRAINT_ERROR` when evaluating loops of the form:

```
for I in INTEGER'LAST-10 .. INTEGER'LAST loop ...
for I in reverse INTEGER'FIRST .. INTEGER'FIRST + 10 loop ...
```

When evaluating null ranges, care must also be taken if the absolute difference between the bounds can exceed `INTEGER'LAST`, e.g.,

```
for I in INTEGER'LAST .. INTEGER'FIRST loop ...
```

S6. Note that an enumeration type can be given a representation such that successive components of the type do not have successive integer values, e.g.,


```

type E is (A, B, C, D);
for E use (3, 10, 50, 1000);
...

```

```

for J in A .. C loop

```

J must take on the values A, B, C successively. In effect, an implementation could implement such a loop as

```

for J' in E'POS(A) .. E'POS(C) loop

```

and then wherever J was mentioned in the original loop, the implementation would supply E'VAL(J').

S7. Note that the loop parameter specification is the declaration of the loop_parameter. According to RM 8.3/5 and RM 8.3/18, a loop parameter is only visible after its declaration. In addition, since the scope of the loop parameter starts at the beginning of its declaration (RM 8.2/2), i.e., from the beginning of the loop parameter specification, any outer declaration of I is hidden (RM 8.3/15). Hence, the loop parameter cannot be named in its own specification:

```

declare
  I : INTEGER := 0
begin
  for I in 1 .. I loop ... end loop;      -- illegal
end;

```

is illegal. The following is also illegal:

```

L:  for I in 1..L.I loop ... end loop;    -- illegal

```

Changes from July 1982

S8. There are no significant changes.

Changes from July 1980

S9. There are no significant changes.

Legality Rules

See also IG 3.6.1/L.

- L1. The loop_parameter must not appear in an assignment context, i.e., as the target of an assignment statement or as an in out or out parameter of a subprogram or entry, or as an in out parameter of a generic instantiation.
- L2. A primary or a function name in an expression of a loop parameter specification cannot be a simple name that is the same as the loop parameter nor can it be an expanded name whose selector is the loop parameter and whose prefix denotes the loop.

Test Objectives and Design Guidelines

The tests for discrete_range (IG 3.6.1/T) are repeated here to ensure that discrete_ranges used in loops are processed correctly.

- T1. Check that a loop_parameter cannot be used as the target of an assignment statement or as an actual in out or out parameter.

Implementation Guideline: Include cases where the loop_parameter is a parameter in procedure calls, entry calls, and generic instantiations.

- T2. Check that if a `loop_parameter` is renamed, the new name cannot be used as the target of an assignment statement or as an actual in out or out parameter (see IG 8.5/T4).

Implementation Guideline: Include cases where the `loop_parameter` is a parameter in procedure calls, entry calls, and generic instantiations.

- T3. Check that the `loop_parameter` is assigned values in ascending order if `reverse` is absent, and descending order if `reverse` is present. (Note: loops over enumeration types with user-defined representations are tested in IG 5.5.b/T16.)

- T4. Check that the loop is not entered if the lower bound of the discrete range is greater than the upper bound, whether or not `reverse` is present.

Check that the loop bounds are evaluated only once, upon entry into the loop.

Implementation Guideline: Use both static and dynamic bounds. At least one test should specify bounds with fairly complex arithmetic expressions. Attempt to modify the loop bounds by changing the value of a variable or subscript used in the `discrete_range`.

- T5. Check that loops whose upper bound is `INTEGER'LAST` (for non-`reverse` loops) and whose lower bound is `INTEGER'FIRST` (for `reverse` loops) are executed without raising `NUMERIC_ERROR` or `CONSTRAINT_ERROR`.

- T6. Check that loops can be specified for `BOOLEAN`, `CHARACTER`, `INTEGER`, user-defined enumeration types, and types derived from these types. Use all four forms of `discrete_range` (`ST`, `L..R`, `ST range L..R`, and `A'RANGE`). Include types derived from derived types.

- T7. If an implementation supports `LONG_INTEGER` or `SHORT_INTEGER`, check that loops using literals and variables of these types can be written.

Implementation Guideline: Note that in these cases the type of the literal must be indicated explicitly by writing one of the following forms:

```
for I in SHORT_INTEGER range 1 .. 10 loop
for I in 1 .. SHORT_INTEGER' (10) loop
```

- T8. Check that integer literals outside the range of `INTEGER` raise `NUMERIC_ERROR` when used in a loop of the form `for I in L .. R loop` (see IG 4.6/S).

- T9. Check that the type of a `loop_parameter` is correctly determined. In particular, for a loop of the form

```
for I in L .. R loop
```

where `L` and `R` are integer literals, check that `I` has the type `INTEGER` and cannot be used in a context requiring a value of a type derived from `INTEGER` or requiring a `LONG_INTEGER` or `SHORT_INTEGER` value.

Implementation Guideline: Separate tests should be written for `LONG_INTEGER` and `SHORT_INTEGER`.

- T10. Check that if `L` or `R` are overloaded enumeration literals, the overloading is properly resolved and the `loop_parameter` is considered to have the appropriate type (see Overloading Resolution in IG 3.6.1.a/S).

- T11. Check that the type of a `loop_parameter` is correctly determined for loops of the form:

```
for I in ST range L .. R loop
```

In particular, if `ST` is a subtype of `T`, check that `I` can be assigned to variables declared with some other subtype of `T` as well as to variables of type `T`.

Check that the above form is accepted even if `L` and `R` are both overloaded enumeration literals (so `L .. R` would be ambiguous if `ST` were omitted).

- T12. Check that the subtype of a loop_parameter is correctly determined so that when the loop_parameter is used in a case statement, an others alternative is not required if the choices cover the appropriate range of subtype values. Use loops of the form

```
for I in ST range L .. R loop
```

where

- L and R are both static expressions, and ST is a static subtype covering a range greater than L .. R. (The case statement alternatives must only cover the range L .. R);
- L or R is a nonstatic expression, but ST is a static subtype. (The case statement must cover the range ST'FIRST .. ST'LAST, ST'BASE'FIRST .. ST'BASE'LAST.)
Implementation Guideline: Check that it is illegal for case alternatives to cover just the range L .. ST'BASE'LAST when L is static, or ST'BASE'FIRST .. R when R is static.
- L and R are static expressions but ST was defined with a nonstatic bound. (The case statement must cover the range ST'BASE'FIRST .. ST'BASE'LAST.)

- T13. Using a case statement, check that in loops of the form:

```
for I in L .. R loop
```

the subtype of I is equivalent to that associated with a loop of the form:

```
for I in T range L .. R loop
```

where T is INTEGER if L and R are integer literal expressions.

Implementation Guideline: Use both integer expressions and enumeration literals for L and R.

- T14. Using a case statement, check that in loops of the form:

```
for I in ST loop
for I in A' RANGE loop
```

the subtype of I is ST'FIRST .. ST'LAST only when ST is static and for A'RANGE, the full range of A's index base type must be covered (since A'RANGE is never static) (see IG 4.9/S).

Implementation Guideline: Use A'RANGE for multidimensional arrays as well as single dimension arrays.

- T15. Check that if a discrete range of the form ST range L .. R raises an exception because L or R is a nonstatic expression whose value is outside ST's range of values, control does not enter the loop before the exception is raised.
- T16. Check for correct processing of iterations over an enumeration type whose representation is user-defined as a non-contiguous set of integers. Use both reverse and normal loops.
Implementation Guideline: Use UNCHECKED_CONVERSION to check that the proper representations are assigned to the loop_parameter.
- T17. Check that a loop_parameter cannot be a subscripted variable, a record component, or a selected name identifying a variable in the scope enclosing the loop.
- T18. Check that the loop_parameter cannot be used in the definition of the discrete range of the iteration rule.
Implementation Guideline: Use both a simple name and an expanded name.

5.5.c WHILE Loops

Semantic Ramifications

S1. Note that although

for I in ST range L.. R loop

is allowed,

while I in ST range L..R loop

is not allowed, since a membership test requires a type mark, not a subtype indication, after in.

Changes from July 1982

S2. There are no significant changes.

Changes from July 1980

S3. There are no significant changes.

Legality Rules

L1. The expression following **while** must have a boolean type (i.e., a type derived from **BOOLEAN** is allowed).

Test Objectives and Design Guidelines

T1. Check that the expression following **while** can have a type derived from **BOOLEAN**.

T2. Check that if the **while** expression is statically or nonstatically **FALSE** when the iteration-specification is evaluated, the loop is not entered.

Check that the **while** condition is evaluated before each iteration, e.g., if an assignment changes the value of the **while** expression, the loop is exited the next time the condition is evaluated.

Implementation Guideline: Change the value of the condition by assigning to a variable used in the expression and by changing the value of a subscript used in the expression.

T3. Check that the **while** expression may be arbitrarily complicated for both static and nonstatic expressions.

5.5.d Continuous Loops

Semantic Ramifications

Changes from July 1982

S1. There are no significant changes.

Changes from July 1980

S2. There are no significant changes.

Test Objectives and Design Guidelines

T1. Check that **loop .. end loop** is executed until control is explicitly transferred out of the loop (via an **exit**, **goto**, or **return** statement) or implicitly by an exception.

5.6 Blocks

Semantic Ramifications

The declarative part of a block is tested in IG 3.9/T. The exception handling part of a block is tested in IG 11/T. The fact that each of these parts is optional in the presence of the other is also tested in IG 11/T. The fact that both these parts may be missing at the same time, leaving the block in the simple form

```
begin
    <sequence_of_statements>
end
```

will be tested here.

Changes from July 1982

S1. There are no changes.

Changes from July 1980

S2. The exception handling portion of the block statement can no longer be empty.

Legality Rules

- L1. If a block name is given, the identifier in the end statement closing the block must be present and must be the same simple name.
- L2. There must be at least one statement between the begin and end parts of the block statement.
- L3. There must be at least one exception handler between the exception and end parts of the block statement if exception is present.
- L4. Within the sequence_of_statements and the (optional) exception handling part of a subprogram body, package body, task body, or generic unit (and excluding any nested subprograms, packages, tasks, or generic units), no identifiers used for statement labels, block identifiers, or loop identifiers are allowed to be the same (RM 5.1/4).

Test Objectives and Design Guidelines

Transfers of control within a block and out of a block are tested in IG 5.9/T2, /T3. The detection of (illegal) transfers of control into blocks is tested in IG 5.9/T1.

Redeclaration of identifiers and name identification issues are treated in IG 8.3/T.

T1. Check that:

- a named block must be closed by an end statement which repeats the block identifier;
- more than one identifier cannot be given as the name of a block;
- the declarative part of a block may be empty, as in:

```
declare
begin
    sequence_of_statements
exception
    exception_handler
end;
```

but the `sequence_of_statements` and the `exception_handler` must not be empty. (The `sequence_of_statements` may consist of nothing but the "empty" statement `null`.) Note: the requirement for a null statement is checked in IG 5.1/T3.

- the following

```
declare
exception
  when ... => ...;
end;
```

is forbidden;

- blocks can be embedded in blocks;

Implementation Guideline: As a capacity test, check at least 65 levels of nesting. There is no need to have declarative parts in these blocks.

- the syntax used for blocks in other block-structured languages

```
begin
  declarative_part
  sequence_of_statements
end
```

is not valid in Ada (see IG 5.1/T1).

- T2. Check that blocks can have declarative parts, and that the declarations in these parts have an effect limited to the block in which they appear.

5.7 Exit Statements

Semantic Ramifications

- S1. The following construct is forbidden:

```
A:  loop
      declare
        procedure P is
        begin
          loop
            exit A;  -- illegal
          end loop;
        end P;
      begin
        null;
      end;
    end loop;
```

Note that the outer loop name is visible within `P`, but since the loop being exited, i.e., `A`, is outside the procedure enclosing the exit statement, the exit statement is illegal (RM 5.7/3, last sentence). Hence, it is not sufficient just to check that an exit statement is inside a loop; it must also be checked that the effect of the exit statement would not be to transfer control out of a subprogram, package, task, or accept statement.

- S2. Note that when leaving a block via an exit statement, a check must be made to ensure that any tasks dependent on the block are terminated (see RM 9.4 and IG 9.4/S):

```

loop
  declare
    task T;
    task body T is ... end;
  begin
    exit;      -- no control transfer until T terminates
  end;
end loop;

```

Changes from July 1982

S3. There are no significant changes.

Changes from July 1980

S4. There are no significant changes.

Legality Rules

- L1. The loop_name in an exit statement must be the name of an enclosing loop statement.
- L2. The loop to which an exit statement applies must not be outside a subprogram body, package body, task body, generic unit body, or accept statement that also contains the exit statement.
- L3. The expression following when must have a BOOLEAN type (i.e., a type derived from BOOLEAN is allowed.)

Test Objectives and Design Guidelines

T1. Check that:

- exit statements cannot be written outside a loop body.
- an exit statement cannot transfer control out of a subprogram, package, task, or accept statement.
Implementation Guideline: These tests should be performed inside a block wholly contained in the body of a loop.
Implementation Guideline: Include generic units.
- the loop parameter cannot be used as a loop_name in an exit statement.
- an exit statement with a loop_name must be enclosed by a loop statement with the same name.

T2. Check that a simple exit unconditionally transfers control out of the innermost enclosing loop. Three contexts should be considered -- the statement is:

1. one of the simple statements in the sequence_of_statements constituting the body of the loop;
2. inside a block in the body of the loop; exits should be attempted from the body of the block and from an exception handler for the block (see IG 11.2/T7 for the exception handler test);
3. inside a compound statement in the body of the loop.

T3. Check that the exit statement condition is evaluated each time through a loop, and that it is evaluated correctly whether positioned at the beginning, middle, or end of the loop.

- T4. Check that an exit statement with a loop_name terminates execution of the enclosing loop statement with the same name, as well as all inner enclosing loop statements.
- T5. Check that conditions of a type derived from BOOLEAN are allowed in exit statements.

5.8 Return Statements

Semantic Ramifications

S1. A subprogram's exception handler may contain a return statement that returns from the subprogram (see RM 11.2/7).

S2. The restriction on where return statements can appear boils down to forbidding return statements in package, generic package, or task bodies. However, the restriction must be carefully checked to ensure that the following examples are illegal:

```

procedure P is
  package Q is ... end;
  package body Q is
  begin
    return;      -- illegal
  end;
begin
  declare
    package QQ is ... end;
    package body QQ is
    begin
      return;    -- still illegal
    end;
  begin
    ...
  end;
end P;

```

The above examples, of course, are illegal even if the return statement is nested in the statement part of a block inside the package bodies. So the relevant consideration is to ensure that the *innermost* construct enclosing a return statement is the **begin-end** portion of a subprogram body (generic or nongeneric), not the **begin-end** portion of a package (generic or nongeneric) or task body.

S3. CONSTRAINT_ERROR is raised by a return statement under the same conditions that this exception would be raised when passing the value to a parameter having the subtype of the function. In particular, if the function's subtype is a constrained array type, the returned value must have the bounds of the subtype:

```

subtype STR2_4 is STRING(2..4);
function F return STR2_4
begin
  return ("abc");      -- CONSTRAINT_ERROR: bounds are 1..3
end F;

```

In particular, returning a value is not semantically equivalent to assigning the value to a variable having the function's subtype:

```

X : STR2_4 := ("abc");      -- no CONSTRAINT_ERROR

```


(Note: no `CONSTRAINT_ERROR` would be raised by `return "abc"`, since the bounds of the string literal would be determined by the index constraint defined by `STR2_4` (RM 4.3.2/9).)

Changes from July 1982

S4. There are no significant changes.

Changes from July 1980

S5. There are no significant changes.

Legality Rules

- L1. The innermost construct enclosing a `return_statement` must be the `begin-end` portion of a subprogram or a generic subprogram, or the `do-end` portion of an `accept statement`, not the `begin-end` portion of a package body, generic package body, or task body.
- L2. `Return_statements` in functions (generic and nongeneric) must have expressions. `Return_statements` in procedures (generic and nongeneric) and `accept statements` must not have expressions.
- L3. The base type of a `return_statement`'s expression must match the base type of the type specified in the function's specification or declaration. (Note: the subtypes do not have to match.)

Exception Conditions

- E1. `CONSTRAINT_ERROR` is raised if a function's subtype is not satisfied by the value of the `return statement`'s expression. In particular, `CONSTRAINT_ERROR` is raised if the function's subtype is:

- a scalar type and the return value is not in the range specified for the subtype.
- a constrained array type and the bounds of the return value do not equal the bounds specified for the subtype.
- a constrained type with discriminants and the discriminants of the return value do not equal the discriminant values specified for the subtype.
- a constrained access type, the return value is not null, and the designated object's bounds or discriminants do not equal the corresponding values for the subtype.

Test Objectives and Design Guidelines

- T1. Check that a `return_statement` cannot appear outside the body of a subprogram, a generic subprogram, or an `accept_statement`.

Implementation Guideline: After trying simple cases, try to return from a package body and a task body where the package or task is nested in a block contained in a subprogram. Also try putting the package in the declarative part of a subprogram.

Check that a `return statement` is permitted in an exception handler (see IG 11.2/T5).

- T2. Check that `return_statements` in functions and generic functions must have expressions.

Check that `return_statements` in procedures, generic procedures, and `accept statements` cannot have expressions.

Implementation Guideline: Use procedures nested in functions and vice versa.

- T3. Check that the base type of a `return_statement`'s expression must match the base type of the type specified in a function's specification or declaration.

Implementation Guideline: The subtype of the return expression should be different from the subtype of the function and still be legal.

- T4. Check that a `return_statement` completes execution of the innermost enclosing subprogram, generic subprogram instantiation, or `accept_statement`.

Check that for functions and generic functions, the value specified in the `return_statement` is actually returned.

Implementation Guideline: Check these for both recursive and nonrecursive subprograms. (Return statements in exception handlers are checked in IG 11.2/T5.)

- T5. Check that the constraints on the return value of a function and generic function are satisfied (or `CONSTRAINT_ERROR` is raised) when the function returns control to its invoker.

Implementation Guideline: Check for integer, enumeration, floating, fixed, constrained array (limited and unlimited), constrained record (limited and unlimited), constrained private, constrained limited private, and constrained access types (with both array, record, private, and limited types as the designated type).

Implementation Guideline: For the array subtype case, be sure the returned value has the correct number of components so `CONSTRAINT_ERROR` is raised only because the bounds differ. Try a null array also.

Implementation Guideline: For the access subtype case, check that `CONSTRAINT_ERROR` is not raised when the returned value is null.

Implementation Guideline: Use a subtype name in the function specification or declaration that is more restrictive than the subtype name for a variable used as the return statement expression.

Implementation Guideline: Try some subtests where the return statement's expression value satisfies the function return value subtype constraints (which must return control normally), and some subtests where the constraints are not satisfied (which must raise an appropriate exception; see IG 5.8/E1).

- T6. Check that if the evaluation of a return statement's expression raises an exception, it can be handled within the body of the function or generic function without necessarily propagating the exception to the function's invoker.

5.9 Goto Statements

Semantic Ramifications

S1. The first sentence of RM 5.9/3 implies that a goto statement cannot transfer control between the alternatives of a case statement, if statement, or select statement since the structure of these statements is:

```

if condition then
    sequence_of_statements
else
    sequence_of_statements
end if;

```

Hence, an if statement like the following:

```

if BE then
    <<L>> null;      -- (1)
else
    goto L;           -- (2)
end if;

```

is illegal, the innermost `sequence_of_statements` enclosing the target statement (1) does not also enclose the goto statement. Note that the RM rule does not prohibit transfers of control from inner statements to outer statements:

```

if BE then
    <<L>>
    ...

```

```

        if BE then
            goto L;    -- legal
        end if;
    end if;

```

Here the `sequence_of_statements` enclosing L also encloses the `goto` statement.

S2. The net effect of the RM rule is to prohibit transfers of control:

- from outside a compound statement (i.e., an if, loop, case, accept, or select statement, or a block) or exception handler into the compound statement or handler;
- from one alternative of an if statement, case statement, or select statement to another alternative of the same statement;
- from one exception handler to another handler of the same unit;
- from an exception handler into the statements of the block, subprogram body, package body, or task body associated with the handler;
- into a subprogram, package, task, or generic unit (such transfers are also prohibited by the rules defining the visibility of labels; see RM 5.1/3).

S3. The first part of the second sentence of RM 5.9/3 prohibits using a `goto` statement to transfer out of an accept statement or a subprogram, package, or task.

S4. Although a `goto` statement cannot transfer control from outside into a subprogram, package, task, exception handler, or compound statement (i.e., an if, loop, case, accept, or select statement, or a block) this restriction does not necessarily imply that these constructs form name scopes for labels (see IG 8.3.a/S).

S5. Note that when leaving a block via a `goto` statement, a check must be made to ensure that any tasks dependent on the block are terminated (see RM 9.4 and IG 9.4/S):

```

declare
    task T;
    task body T is ... end;
begin
    loop
        goto L;    -- no transfer until T terminates
    end loop;
end;

```

Changes from July 1982

S6. There are no changes.

Changes from July 1980

S7. There are no significant changes.

Legality Rules

- L1. A `goto` statement cannot attempt to transfer control out of an accept statement or a subprogram, package, task, or generic unit.
- L2. A `goto` statement cannot attempt to transfer control into a compound statement (i.e., an if, loop, case, accept, or select statement, or a block), subprogram, package, task, exception handler, or generic unit.

- L3. A goto statement cannot attempt to transfer control between the alternatives of a case statement, if statement, or select statement, or between statements belonging to different exception handlers in a sequence of handlers.
- L4. A goto statement in an exception handler cannot transfer control into the sequence of statements guarded by the set of handlers.

Test Objectives and Design Guidelines

T1. Check that

- a. a goto statement cannot reference nonexistent labels;
- * b. a statement with multiple labels can be a target of a goto statement, and any one of its labels may be used for this purpose;
- c. a goto statement cannot transfer control into or out of a subprogram, package, task, accept statement, or generic unit;
- d. a goto statement cannot transfer control into an if, loop, case, accept, or select statement, or a block;
- e. jumps between alternatives of a case statement, if statement, or select statement are not permitted;
- f. jumps from an exception handler in a unit to another belonging to the same unit are not allowed;
- g. jumps out of an exception handler to a label declared in the statements guarded by the exception handling part of a block, subprogram body, task body, or package body are not permitted;
- h. jumps into an exception handler from outside the handler are not allowed.

- T2. Check that jumps out of an exception handler to a statement in an enclosing unit are allowed and are performed correctly.

Check that jumps out of compound statements (other than an accept statement) are allowed and are performed correctly.

Implementation Guideline: Jump to statements which precede and follow the compound statement.

Check that jumps out of select statements (other than from inside accept bodies in select alternatives) are possible and are correctly performed.

- T3. Check that goto statements correctly transfer program control. (This objective is implicitly checked by other tests in this section.).

Chapter 6

Subprograms

6.1 Subprogram Declarations

Semantic Ramifications

S1. RM 2.3/3 says that the case of letters used in any identifier (including an operator) is not significant. Hence, the specification of operators in RM 4.5/2 implicitly includes all forms with upper case letters, which means that the case of the letters in a string literal representing an operator is not significant. Similarly, since an identifier does not syntactically include any leading or trailing spaces, the string literal representing an operator cannot include such spaces. Finally, no embedded spaces are allowed since the only operators having embedded spaces are the short-circuit control forms and then and or else, and the membership tests in and not in. None of these are overloadable operators (RM 4.5/1). Consequently, no spaces are allowed within an operator symbol.

S2. Note that a default value can be provided for a formal parameter of an unconstrained array type or of an unconstrained type with discriminants that do not have default values. In this respect, a parameter declaration is similar to a constant declaration. (However, see IG 6.4.2/S for some important differences.)

S3. Although a parameter declaration may not use a name that refers to a parameter declared earlier in the same formal part, it may use the name of a parameter declared later in that part if such a name has a meaning outside the parameter declaration, e.g.,

```
C : constant INTEGER := 5;
type T is new INTEGER;
procedure P (A : T;                -- (1)
            T : INTEGER := C;      -- (2)
            C : INTEGER);          -- (3)
```

Type T and constant C can be referred to by their simple names until their declarations are hidden. Type T is not hidden until the beginning of parameter T's declaration. Similarly, constant C is not hidden until the beginning of parameter C's declaration (RM 8.3/15 and RM 8.2/2). Note that C at (2) does not refer to the formal parameter declared in P's formal part; it refers to the constant C, and hence, the rule in RM 6.1/5 does not apply. The rule in RM 6.1/5 forbids a declaration like the following:

```
procedure Q (A : INTEGER; B : INTEGER := A);
```

This rule is needed since parameter A is visible within B's declaration (RM 8.3/5).

S4. A parameter declaration may also use the identifier of a parameter declared earlier in the formal part as long as the identifier does not refer to the formal parameter, i.e., the identifier may be used as a selector in a component selection, as a component simple name in an aggregate, or as a parameter name in a named parameter association:

```
package P is
  type F is
    record
      F : INTEGER;
    end record;
  subtype FF is F;
```

```

G : F;
function FUNC (F : INTEGER) return INTEGER;

procedure Q (F : INTEGER;
             A : P...           -- legal use of F
             B : FF := (F => 3); -- legal use of F
             I : INTEGER := FUNC(F => 3); -- legal use of F
             J : INTEGER := G.F); -- legal use of F
end P;

```

S5. A parameter declaration may, of course, use the value of a formal parameter of an enclosing subprogram, e.g.,

```

subtype SMALL is INTEGER range 1 .. 100;
procedure P1 (X, Y : INTEGER) is
begin
  procedure P2 (Z : SMALL := X);

```

Calls to P2 will raise `CONSTRAINT_ERROR` if the default is used and X is not in the range 1 .. 100.

S6. The elaboration of a subprogram specification cannot raise an exception (see IG 11.4/S).

S7. In an explicit declaration of a subprogram, entry, or generic subprogram, a formal parameter of mode out may have a limited type only under the following circumstances (RM 7.4.4/4):

- the type is a limited *private* type (i.e., not a composite limited type nor a task type);
- the subprogram, generic subprogram, or entry declaration occurs within the visible part of the package that declares the limited private type;
- the full declaration of the limited private type does not itself declare a limited type.

One of the consequences of this rule is that a task type cannot ever be used as the type of an out parameter, nor can a limited private type whose full declaration declares a task type. Another consequence is that, even though subprograms having parameters of limited types may be declared outside the package that declares the type, such subprograms may not have an out parameter of that type.

S8. The restriction forbidding subprogram out parameters of limited types applies only to "explicit subprogram declarations," entry declarations, and generic procedure declarations. Since a generic formal subprogram is not declared with a subprogram declaration, this restriction does not apply to formal subprograms, nor does it apply to renaming declarations.

```

generic
  type LP is limited private;
  with procedure P (X : out LP); -- legal
package P1 is
  procedure S (X : out LP); -- illegal
  procedure NP (X : out LP) renames P; -- legal
end P1;

```

S9. Within the specification of a subprogram, every declaration with visibility higher than the subprogram is hidden (RM 8.3/16). This rule means that no subprogram can be declared to a

subprogram simple name can be used within the subprogram specification as the selector in a component selection, as a component simple name in an aggregate, or as a parameter name in a named parameter association:

```

package P is
  type F is
    record
      F : INTEGER;
    end record;
  subtype FF is F;
  G : F;
  function FUNC (F : INTEGER) return INTEGER;

  procedure F (A : P.F;
               B : P.FF;
               F : INTEGER;
               H : FF := (F => 3);
               I : INTEGER := FUNC(F => 3);
               J : INTEGER := G.F);
end P;

```

-- illegal use of F
 -- ok
 -- ok use of F
 -- illegal use of F
 -- illegal use of F
 -- illegal use of F

Note that such examples cannot be written for operator symbols, since an operator symbol can never denote a type, a formal parameter, or a record component. Moreover, an operator symbol can never be used in a default expression of a function whose designator is an operator symbol, since default expressions are forbidden for such functions (RM 6.7/2).

S10. Since no rule forbids a subprogram calling itself or being called from different tasks, all subprograms can be called recursively and are reentrant.

Changes from July 1982

S11. There are no significant changes.

Changes from July 1980

S12. The return type in a function specification must be a type mark; no *explicit* constraint is allowed.

S13. The type of a formal parameter must be specified by a type mark; no *explicit* constraint is allowed.

S14. Default expressions are not evaluated until the subprogram call (see RM 6.4.2).

S15. Names of variables, calls to user-defined operators, calls to functions, and allocators may now appear in the default expression of a formal parameter.

Legality Rules

- L1. An operator_symbol used as a designator in a subprogram specification must not be one of the following strings: "/"=" (RM 6.7/4) or "ln" (RM 6.7/4), nor may the literal contain spaces.
- L2. A default expression is allowed only for formal parameters with mode in and only for subprograms not designated with operator symbols (RM 6.7/2).
- L3. The base type of a default expression must be the same as the base type of its formal parameter.
- L4. A simple name is not allowed in a parameter declaration if the name refers to a formal parameter declared earlier in the same formal part.

- L5. The simple name of a subprogram cannot be used within the subprogram's formal part except to declare a formal parameter having the name of the subprogram (RM 8.3/16). In particular, the subprogram's name cannot be used as a selector in a component selection, as a component simple name in an aggregate, as a parameter name in a named parameter association, or as a simple name in a default expression.
- L6. An explicit declaration of a subprogram (in a subprogram declaration, generic instantiation, renaming declaration, or formal generic subprogram declaration) must not be a homograph of another declaration occurring immediately within the same declarative region unless exactly one of these declarations is the implicit declaration of a predefined operation, or exactly one of them is the implicit declaration of a derived subprogram (RM 8.3/17) (see also IG 6.6-S).
- L7. Formal parameter identifiers of subprograms must be distinct from each other and from identifiers declared in the subprogram's declarative part or a preceding generic part (RM 8.3/17).
- L8. A function subprogram must only have parameters of mode in (RM 6.5/1).
- L9. An out parameter of a subprogram, generic subprogram, or entry declaration must not have a limited type unless (RM 7.4.4/4):
- the type is a limited private type,
 - the declaration of the subprogram, generic subprogram, or entry occurs within the visible part of the package that declares the limited private type (including within any nested packages), and
 - the declaration of the limited private type does not declare a limited type.
- L10. The operator symbols "and", "or", "xor", "=", "<", "<=", ">", ">=", "&", "*", "/", "mod", "rem", and "/" must only be used in function specifications having two parameters (RM 6.7/2). (Strings with upper case letters replacing lower case letters are also allowed.)
- L11. The operator symbols "+" and "-" must only be used in function specifications having one or two parameters (RM 6.7/3).
- L12. The operator symbols "not" and "abs" must only be used in a function specification having a single parameter (RM 6.7/2). (Strings with upper case letters replacing lower case letters are also allowed.)
- L13. Formal parameters for the "=" operator must have the same type, and except in renaming declarations, the type must be a limited type (RM 6.7/4).
- L14. If a declaration and body (or body stub) are both given for a subprogram, the body must appear after the declaration (RM 3.9/9).

Test Objectives and Design Guidelines

- T1. Check that certain syntactic malformations are forbidden, viz.:
- a subprogram cannot be declared as an object, a type, or a formal parameter;
 - the identifier **returns** cannot be used in place of **return** in a function specification;
 - the reserved word **return** cannot be used in the specification for a procedure;
 - the operator_symbol for a one-character operator cannot be written as a character literal, e.g., '+';

- parameter declarations cannot be separated with commas;
- the mode designation out in is forbidden;
- a semicolon is not allowed in place of is for a procedure or function;
- a formal part for a function or procedure cannot have the form "()";
- an array type definition is forbidden in a formal parameter declaration;
- the type of a formal parameter must be designated by a type mark;
- the return type of a function must be designated by a type mark;
- default expressions are not allowed for functions declared as operator symbols (see IG 6.7/T1).

- T2. Check that "in", "not in", "and then", "or else", "/=", and "!=" are not permitted as operator_symbols in subprogram declarations (see IG 6.7/T1).

Check that leading or trailing blanks are not allowed in operator symbols (see IG 6.7/T1).

Check that all the permitted operator_symbols can be used in function specifications with the required number of parameters (see IG 6.7/T2).

Check that when the permitted operator symbols are used in function specifications, the case of the letters in the string literals is not significant (see IG 6.7/T2).

Check that functions for "and", "or", "xor", "=", "<=", "<", ">=", ">", "&", "*", "/", "mod", "rem", and "" cannot be declared with one or three parameters (see IG 6.7/T1).

Check that functions for "not" and "abs" cannot be declared with two parameters (see IG 6.7/T1).

Check that functions for "+" and "-" cannot be declared with zero or with three parameters (see IG 6.7/T1).

- T3. Check that duplicate subprogram specifications are not allowed in the same declarative region (see IG 6.6/T1).

Check that a subprogram declaration and subprogram body can be given separately in the same declarative part, but that the subprogram declaration must precede the subprogram body.

- T4. Check that duplicate formal parameter names are forbidden in a single formal_part, and that a formal parameter, a generic parameter, and a local variable or enumeration literal cannot have the same name (see IG 8.3.e/T1).

- T5. Check that default expressions are forbidden for formal parameters of mode in out or out.

- T6. Check that the type of a default expression must be the same as the base type of the formal parameter.

- T7. Check that an unconstrained record type (with and without default constraint values) and an unconstrained array type are permitted as formal parameter types (see IG 6.4.1/T6).

- T8. Check that CONSTRAINT_ERROR is not raised when a subprogram is declared if the value of the default expression for the formal parameter does not satisfy the constraints of the type mark, but is raised when the subprogram is called and the default value is used.

Implementation Guideline: Try an array parameter constrained with nonstatic bounds and initialized with a static aggregate, a scalar parameter with nonstatic range constraints initialized with a static value, and a record parameter whose components have nonstatic constraints initialized with a static aggregate.

- T9. Check that names of variables, calls to user-defined operators, and in turn forms, and allocators may be used in default expressions for formal parameters.
- T10. Check that a formal parameter of mode in and in out cannot be of a task type, including a composite type.
- Check that a formal parameter of mode out may be a pointer to a task type (see IG 7.4.4.7).
- Check that a formal parameter of mode out cannot be a task type (see IG 7.4.4.7).
- T11. Check that a name referring to a formal parameter cannot be used outside the same formal part, and that a parameter's identifier can be used if it does not refer to the parent object.
- T12. Check that the identifier of a subprogram cannot be used within its own part as a selector, as a component simple name in an aggregate, as a selector in a task, or as a named association, or as a simple name in a default expression (see also IG 7.4.4.7).
- T13. Check that subprograms can be called recursively and reentrantly (see IG 7.4.4.7).
- Check that local variables, local variables, and parameters of containing subprograms can be accessed directly from within a recursive call (see IG 7.4.4.7).

6.2 Formal Parameters

Semantic rules for errors

- S1. If the mode of a formal parameter is an access type, then no assignments can be made to any subcomponent of the parameter's designated object, no discriminants of the designated object can be assigned, and no attributes can be applied to the parameter object. RM 4.1.4 says:
- "If the mode of a parameter is an access type, then the prefix may not be a name that denotes a component of a task, a task object, a task object of mode out or a subcomponent thereof."

Consequently, the following forms of name are forbidden on the parameter of a task or AR are formal objects:

- AR.C -- indexed component, when AR is a task object or a task object of mode out or a slice, when AC designates an array
- AR.C.D -- selected component, when AR designates a record
- AR.C.D.E -- attribute, when AR has an attribute designated by E, even when the designated object is an access type
- AR.C.D.E.F -- indexed component, when AR is a record and AR.C is a component of type is an access type designating an array
- AR.C.D.E.F.G -- slice, when AR is a record and AR.C is a component of an access type designating an array
- AR.C.D.E.F.G.H -- selected component, when AR is a record and AR.C is a component designating a record
- AR.C.D.E.F.G.H.I -- attribute, when AR.C is a record component having an access type

Note that a formal parameter (except for discriminants and bounded variables) may be assigned the procedure value of a procedure value has been assigned to the parameter by the procedure.

- S2. The parameters FIRST, LAST, LENGTH, and RANGES are of mode out since they read the bounds of the parameter. They cannot be applied to a parameter of an access type even if the designated object is an array type (RM 4.1.4).

S3. RM 3.7.4/3 explicitly permits the attribute 'CONSTRAINED to be applied to an out formal parameter, A, (or a subcomponent of A) if A (or its subcomponent) has discriminants (even though 'CONSTRAINED does not read a discriminant or bound).

S4. Other attributes that can be applied to an out parameter or its subcomponents are the representation attributes 'ADDRESS, 'SIZE, 'POSITION, 'FIRST_BIT, and 'LAST_BIT. These attributes can be applied because evaluation of the prefix of such attributes means determining the entity denoted by the prefix (RM 4.1/9, /10), and such a determination does not require reading the value of the entity. However, even these attributes cannot be applied to out parameters or out parameter subcomponents of an access type.

S5. No other attributes can be applied to an out parameter or its subcomponents, since the only other attributes that can be applied to objects are 'STORAGE_SIZE, 'CALLABLE, and 'TERMINATED. These attributes all can be applied to task objects, but an out parameter and its subcomponents can never denote a task object (see IG 7.4.4/S and RM 6.1/S).

S6. If a formal in parameter is an object of an access type, assignments can still be made to the object designated by the parameter (i.e., to components of the object or to the object itself), but not to the parameter itself.

S7. Scalar subcomponents (other than discriminants) of out parameters that are not updated during a procedure call are undefined upon return from the call, e.g.,

```
type ARR is array (1 .. 2) of BOOLEAN;
AR : ARR := (TRUE, TRUE);
```

```
procedure P (X : out ARR) is
begin
  X (ARR'LAST) := TRUE;
end P;
```

After the call

```
P (AR);
```

AR(1) is undefined (see also IG 6.4.1/S).

S8. Note that a formal parameter must not have mode out if its type is limited unless

- the type is private,
- the type is declared in the same visible part as the subprogram declaration, and
- the full declaration of the private type is not limited.

In particular, a formal out parameter may not have a task type or a composite type that contains a task type. (See RM 7.4.4/4).

S9. No subcomponent of an out parameter may be used as an in out actual parameter of a subprogram call (RM 6.4.1/3), an entry call (RM 9.5/2, RM 6.4.1/3), or a generic instantiation (RM 12.3.1/2). In addition, no subcomponent of an out parameter (other than discriminant subcomponents) may be used as an in parameter of a subprogram call, an entry call, or a generic instantiation, since such actual parameters are evaluated (i.e., read) when the call is evaluated (RM 6.4.1/2 and RM 12.3/17), and reading of an out formal parameter or its subcomponents (other than a discriminant subcomponent) is not allowed.

S10. A nonerroneous program's semantic effect must be the same whether a parameter is passed by copy or by reference. The effect of the rules for the erroneous use of formal parameters is that an implementation can assume, for purposes of optimization, that an

assignment to any formal parameter will not affect the value of any other formal parameter, nor can any assignment to a global variable affect the value of a formal parameter. Formal parameters are passed by reference, and an assignment to a formal parameter (which, in turn, *does* affect the value of some (other) formal parameter), then the program is erroneous if any attempt is made to read the value of the changed formal parameter. It is the programmer's responsibility to ensure that a program is not erroneous. Since an implementation is allowed to assume that it is processing nonerroneous programs, it is allowed to assume that there is no semantic error implied by the implementation's choice of a parameter-passing mechanism.

S11. If an actual parameter is a subcomponent that depends on a discriminant of an unconstrained record type, then an evaluation of the name requires checking that the component exists at the current value of the discriminant. This check need only be done once before the call, since the name is only evaluated once (RM 6.4.1.4). An implementation need not check for the continued existence of the component after the call, since it is required to exist if an assignment has been made that changed the discriminant value of the containing variable, and such an assignment makes the call erroneous, i.e., an implementation may assume that such an assignment has not occurred:

```

-- RECORD TYPE
RECORD
  D : INTEGER := 0) IS
  --
  S : STRING(1 .. D);
  CASE D IS
    WHEN 3 => C : INTEGER;
    WHEN OTHERS => null;
  END CASE;
END RECORD;

-- CALL
CALL RECVD (3, S => "ABC", C => 4);

-- BODY
-- (K_IO : in out CHARACTER; X_IO : in out INTEGER) IS
--
--   (2, S => "EF");

-- RETURN
RETURN RECVD(C);

```

The call is erroneous because both actual parameters depend on RECV's discriminant, and the body of the procedure changed the discriminant of RECV. The effect of executing the call will be unpredictable, since the implementation can assign the values of the formal parameters to the locations determined at the time of the call without checking to be sure that these locations are still valid components of the variable. Consequently, inappropriate values might be assigned to these locations, leading to unpredictable effects. It is the programmer's responsibility to write erroneous programs.

S12. For parameters of a scalar type, the difference between mode in out and mode out is that the actual parameter value is not copied to the formal parameter for mode out, nor is the value checked against the constraint of the formal parameter (RM 6.4.1.6). For parameters of an access type, the only difference between modes in out and out is that for mode out, the subtype constraint of the parameter is not checked against the subtype of the formal parameter, even though the value of the actual parameter is copied to the formal parameter (RM 6.4.1/S for further details of this point). For out parameters of composite types, the only difference from in out parameters semantics is that scalar nondiscriminant component values need not be passed to the formal parameter. Hence, an array of access values used as an out parameter

must be passed to the formal parameter, just as if it were an in out parameter. Constraint checks between actual and formal parameters are identical for in out and out parameters (RM 6.4.1/9) having composite types.

Changes from July 1982

S13. The value of an actual out parameter, or an out parameter subcomponent, is copied to the formal parameter at the start of a call if the parameter or the subcomponent has an access type.

S14. Inside the procedure only the bounds and discriminant values of a formal out parameter and its subcomponents may be read.

S15. If copy is used for array or record type parameters of mode out, then copy-in is required for the bounds and the discriminants of the actual parameter and its subcomponents, and for each subcomponent whose type is an access type.

S16. For parameters of a private type (including limited private), parameter passing effects are achieved according to the rule for the corresponding full type.

Changes from July 1980

S17. Except for parameters of mode in having a scalar or an access type, the execution of a subprogram call is erroneous if an actual parameter is a subcomponent that depends on the discriminants of an unconstrained record variable, and the value of the discriminants is changed by the execution.

Legality Rules

- L1. A formal in parameter of a subprogram, a generic subprogram, or an entry (RM 9.5/6) must not be used as an actual in out parameter (RM 6.4.1/3), as an actual out parameter (RM 6.4.1/3), as the target of an assignment statement (RM 5.2/1), or as a generic in out actual parameter (RM 12.3.1/2).
- L2. A formal out parameter or a subcomponent of a formal out parameter must not be used as an actual in out parameter of a subprogram or entry call (RM 6.4.1/3), or as an actual in out parameter in a generic instantiation (RM 12.3.1/2).
- L3. A formal out parameter or a subcomponent of a formal out parameter (other than a discriminant subcomponent) must not be used in an expression (RM 6.2/5), except as the prefix of the attribute ADDRESS, CONSTRAINED, FIRST and FIRST(N) (when the parameter has an array type), FIRST_BIT, LAST and LAST(N) (when the parameter has an array type), LAST_BIT, LENGTH and LENGTH(N) (when the parameter has an array type), POSITION, RANGE and RANGE(N) (when the parameter has an array type), or SIZE.
- L4. No attribute can be applied to an out parameter or a subcomponent of an out parameter if the parameter has an access type (RM 4.1/4).

Test Objectives and Design Guidelines

- T1. Check that a formal in parameter cannot be used as the target of an assignment statement, or as an actual parameter whose mode is in out or out, or as a generic in out actual parameter (see IG 12.3.1/T2).

Implementation Guideline: Use a simple scalar in parameter, an array and a record in parameter (attempt to assign to a component of the parameter or use a component as an out actual parameter), and an access in parameter.

Check that in is optional for in parameters of procedures and functions.

Check that labels and subprograms can neither be declared nor passed as subprogram parameters.

- T2. Check that objects designated by in parameters of access types (including the object selected by .all) can be used as the target of an assignment statement and as an actual parameter of any mode.

- T3. Check that scalar and access parameters are copied for all three modes.

Implementation Guideline: Check this by calling subprograms of the form F(A,A) where the second parameter is an in out or out parameter. Assignments to the second formal parameter should not change the value of the first formal parameter, nor should direct assignments to the actual parameter change the value of the corresponding formal parameter. Check this for both procedures and functions.

Implementation Guideline: Check that if an exception is propagated from a subprogram, the values of the actual scalar parameters are the values at the time of the call, even if assignments were made to the formal parameters before the exception was raised.

Check that a private type whose full type declaration declares a scalar or access type is passed by copy for all modes.

- T4. Check that aliasing is permitted for parameters of composite types, e.g., that a matrix addition procedure can be called with three identical arguments, e.g., MAT_ADD(A,A,A);.

Check that for unconstrained array formal parameters, the bounds of the formal parameter are determined by the bounds of the actual parameter, even if a default value is specified for the formal parameter (see IG 6.4.1/T6).

- T5. Check that discriminant values for record, private, and limited private actual parameters are passed to unconstrained formal parameters, even if a default value is specified for the formal parameter (see IG 6.4.1/T6).

- T6. Check that the discriminants of an out formal parameter and its subcomponents may be read inside a procedure, but not other component values.

Check that an out parameter cannot be passed as an in or in out parameter of a subprogram call, an entry call, or a generic instantiation.

Check that 'FIRST, 'LAST, 'LENGTH, 'RANGE, 'ADDRESS, 'SIZE, 'POSITION, 'FIRST_BIT, and 'LAST_BIT cannot be applied to an out parameter of an access type (nor to an access subcomponent of an out parameter), but are allowed for an in or in out parameter.

Check that 'FIRST, 'LAST, 'LENGTH, 'RANGE, 'ADDRESS, 'SIZE, 'POSITION, 'FIRST_BIT, and 'LAST_BIT can be applied to an out parameter or out parameter subcomponent that does not have an access type.

Check that 'CONSTRAINED is not allowed for a parameter (of any mode) having an access type, even if the designated type is a type with discriminants.

Check that an out parameter or an out parameter subcomponent having an access type cannot be used in a selected component, an indexed component, or a slice.

Implementation Guideline: Check in both expression and assignment contexts.

Check that no out parameter or out parameter subcomponent can be used as an in out actual parameter.

Check that an out parameter can be passed to another out parameter.

- T7. Check that procedure calls are allowed even if they do not assign values to scalar formal out parameters or to scalar components of formal out parameters. No exceptions should be raised, and no errors reported in compilation. (Warnings are allowed, however).

Implementation Guideline: Do not refer to the values after the calls.

- T8. Check that limited formal out parameters are only allowed if the type is private, the private type declaration appears in the same visible part, and the full type declaration of the private type is a nonlimited type (see IG 7.4.4/T1).

- T9. Check that default initialization expressions are not evaluated for out parameters.

6.3 Subprogram Bodies

Semantic Ramifications

S1. A subprogram returns to its caller (RM 6.2/6) when it executes a return statement that does not raise an exception (RM 5.8/6) or when there are no more statements to be executed. Note that such a return occurs for both procedures and functions, although if the last simple statement executed by a function is not a return statement and does not propagate an exception, PROGRAM_ERROR is raised after the return to the caller (see IG 6.5/S).

S2. Since a subprogram body serves to declare a subprogram in the absence of a separate subprogram declaration, when a subprogram body is compiled as a library unit, it serves to declare the subprogram. If later it is necessary to change the subprogram specification --- e.g., by adding an optional parameter --- it is necessary to compile a subprogram declaration to replace the previous specification in the library (see RM 10.1/6):

```

procedure P(X : INTEGER) is          -- first compilation (1)
begin ... end P;

procedure P(X : INTEGER;
            Y : INTEGER := 0) is     -- illegal; does not conform to (1)
begin ... end P;

procedure P(X : INTEGER;
            Y : INTEGER := 0);       -- legal; replaces old declaration

procedure P(X : INTEGER;
            Y : INTEGER := 0) is     -- legal now
begin ... end P;
```

Changes from July 1982

S3. Each subprogram declaration must have a corresponding body, except for subprograms written in another language (see RM 13.9).

Changes from July 1980

S4. The exception part of a subprogram body must contain at least one exception handler.

S5. A body stub can be used to declare a subprogram (in the absence of a subprogram declaration).

S6. See also IG 6.3.1/Changes and IG 6.3.2/Changes.

Legality Rules

- L1. The designator at the end of a subprogram body, if present, must be the same as the designator used in the subprogram specification.
- L2. A subprogram_body is only permitted as a library_unit or in the declarative_part of a package_body, block, subprogram_body, or task_body. (It is not permitted as a declarative_item in a package specification or a task_specification (see RM 7.2 and RM 9.1).)
- L3. The subprogram specification given in the subprogram body must conform to the subprogram specification given in the subprogram declaration (see IG 6.3.1).
- L4. If a declaration and body (or body stub) are both given for a subprogram, the body must appear after the declaration (RM 3.9/9).

Exception Conditions

- E1. **PROGRAM_ERROR** is raised at the point of call if the last simple statement executed within a function body is not a `return` statement and does not propagate an exception (RM 6.5/2).

Test Objectives and Design Guidelines

- T1. Check that the designator at the end of a subprogram body must be the same as the designator used in the subprogram specification.

Implementation Guideline: Use both an operator_symbol and an identifier. Check that selected component notation cannot be used, even if the selected component properly identifies the subprogram.

Implementation Guideline: Check for generic and nongeneric subprograms.

- T2. Check that a subprogram body is forbidden as a declarative item in a package-specification (see IG 7.2/T4) or a task-specification (see IG 9.1/T2).

- T3. Check that a null statement, at least, is required in the body of a subprogram (see IG 5.1/T3).

- T4. Check that a procedure with and without a return statement returns correctly.

- T5. Check that for each subprogram specification there must be a corresponding body (unless the subprogram is written in another language (see IG 13.9/T1)).

Check that a renaming declaration cannot be used to provide a subprogram body.

Implementation Guideline: Check for a subprogram declared in the visible part of a package and a subprogram declared earlier in the same declarative part.

- T6. Check that if a subprogram body has an exception part, at least one handler must be specified.

- T7. Check that an exception raised during the execution of a subprogram body can be handled inside the subprogram body (see IG 11.4/T4).

- T8. Check that recompiling a subprogram library unit body does not replace the subprogram declaration in the library (see IG 10.3/T2).

- T9. Check that the formal part of a subprogram specification in a declaration must conform to the specification given in the body.

Implementation Guideline: Check for generic and nongeneric subprograms.

Implementation Guideline: Check that for parameters of mode `in`, `in` must appear in both formal parts or neither formal part.

Implementation Guideline: Conformance with separately compiled units and body stubs is checked in IG 10.1/T10 and IG 10.2/T11.

- T10. Check that a subprogram body cannot precede its specification in a declarative part (see IG 6.1/T3).

6.3.1 Conformance Rules**Semantic Ramifications**

- S1. Note that:

```
function "MOD" (L, R : T) return T;
function "mod" (L, R : T) return T is ...
```

and


```

procedure P (X : INTEGER := "MOD"(10, 3));
procedure P (X : INTEGER := "mOd"(8#12#, 03)) is ... end P;

```

have conforming formal parts because the string literals denote the same operator and the numeric literals have the same values.

S2. Note that since real literals have type *universal_real*, "have the same value" means have the same *universal_real* value. Hence, it is not sufficient for two literals to map to the same model number. For example, even if the maximum floating point mantissa supported by an implementation is 24 bits, 16#AAA_AAA_A# and 16#AAA_AAA_AA# do not conform, although both might be represented by 16#AAA_AAA#. Similarly, 3#0.1#, 0.33333..., and 16#555_555# do not conform, although they all have approximately the value 1/3.

S3. Since conforming expressions must have the same sequence of lexical elements 3 + 2 does not conform with "+"(3, 2).

S4. A name declared by a renaming declaration or by a subtype declaration does not conform with the name of the renamed entity, since the meaning of the names are given by different declarations:

```

package P is
  X : INTEGER;
  NX : INTEGER renames X;
  procedure Q (A : INTEGER := X);
end P;

package body P is
  procedure Q (A : INTEGER := NX) is      -- illegal
  begin null; end Q;
end P;

```

NX and X are not given the same meaning by the visibility rules (RM 8.3/2) since they are declared by different declarations. Hence the above specifications for Q do not conform according to RM 6.3.1/5.

S5. The visibility rules are applied independently for the names appearing in conforming constructs:

```

N : INTEGER := 5;                                -- (1)
package P is
  procedure R (X : INTEGER := N);
end P;

package body P is
  N : INTEGER := 5;                                -- (2)
  procedure R (X : INTEGER := N) is -- illegal
  ...
end P;

```

The N in the second declaration is associated with the N declared at (2) rather than with the N declared at (1). Hence, the formal parts do not conform.

S6. Since the conformance rules for simple vs. expanded names also require that both names be associated with the same declaration, new and old names introduced by renaming declarations cannot be freely interchanged in the selector of an expanded name:

```

package P is
  type T is new INTEGER;
end P;

with P; use P;
package Q is
  procedure R (X : T);
end Q;

package body Q is
  package S is
    subtype T is P.T;
  end S;
  procedure R (X : S.T) is      -- illegal
  begin ... end;
end Q;

```

Although S.T and P.T denote the same entity, they are not associated with the same declaration; hence, S.T and T do not conform.

S7. New names introduced by renaming declarations can sometimes, but not always, be used in the prefix of an expanded name:

```

package P is
  type T is new INTEGER;
end P;

with P; use P;
package Q is
  procedure R (X : T);
  package QP renames P;
end Q;

```

Now consider the following possible ways of naming P.T and consider which of these pairs satisfy the conformance rules:

```

T, P.T    -- conform
T, QP.T   -- conform
P.T, QP.T -- do not conform

```

T conforms to P.T and QP.T because both selectors denote the same declaration, and RM 6.3.1/3 allows a simple name, T, to be replaced by an expanded name (P.T or QP.T) if the meaning of the selector in the expanded name (i.e., T) is given by the same declaration. However, P.T and QP.T do not conform because we are here replacing one expanded name by another expanded name. Two expanded names conform if their selectors are the same and if their prefixes conform. For P.T and QP.T, the prefix consists of a simple name, but since these simple names are not given by the same declaration, the prefixes P and QP do not conform. Hence, the expanded names P.T and QP.T do not conform. Note that the names STANDARD.P.T and P.T *would* conform since the simple name P in P.T has been replaced by the expanded name STANDARD.P, and both P and STANDARD.P are associated with the same declaration.

S8. The rule specifying when simple names can be "replaced" by expanded names does not specify whether the simple name appears in the first or the second occurrence of conforming constructs. Hence, either can appear first.

S9. The requirement that corresponding lexical elements must denote the same declaration includes operators, of course:

```
package P is
  type T is new INTEGER;
  procedure Q (X : T := 3 + 5);      -- uses predefined "+"
  function "+" (L, R : T) return T;
end P;

package body P is
  procedure Q (X : T := 3 + 5) is    -- illegal; not predefined "+"
  ...
end P;
```

The second use of "+" denotes the user-defined "+" operator, not the implicitly declared predefined "+", so the two specifications of Q do not conform.

S10. RM 6.3.1/3 only allows simple_names, i.e., identifiers (RM 4.1/2), to be replaced by expanded names. Hence, an operator symbol or a character literal cannot be replaced by an expanded name, even if both names denote the same declaration:

```
package P is
  function "+" (L, R : INTEGER) return INTEGER;
  procedure Q (X : INTEGER := "+" (3, 5));
end P;

package body P is
  procedure Q (X : INTEGER := P."+" (3, 5)) is -- illegal
  ...
end P;
```

S11. Note that a pragma is a lexical element; hence, if a pragma appears in the formal part of a subprogram declaration, the same pragma must appear later in the formal part of the subprogram body, even if the pragma is ignored.

Changes from July 1982

S12. A simple name can be replaced by an expanded name if the same declaration gives the simple name and the selector their meanings.

S13. A string literal given as an operator symbol can be replaced by a different string literal if and only if they both denote the same operator.

Changes from July 1980

S14. Literals having the same value are considered to conform.

S15. Conforming constructs must be lexically identical except for comments and special rules regarding literals and expanded names.

Legality Rules

- L1. Conforming constructs must consist of the same sequence of lexical elements except that comments are ignored, certain string literals can be replaced by different string literals (see below), and simple names can be replaced by expanded names if the meaning of both names is given by the same declaration (RM 6.3.1/5).
- L2. A character literal or an operator symbol cannot be replaced by an expanded name denoting the same literal or operator (RM 6.3.1/3).

- L3. Corresponding string literals used as operator symbols can differ only with respect to the case of the letters used in the operator symbol (RM 6.3.1/4).
- L4. Corresponding numeric literals must have the same (*universal_integer* or *universal_real*) value (RM 6.3.1/2).
- L5. Corresponding simple names, character literals, operators, and operator symbols must be declared by the same declaration (RM 6.3.1/5).

Test Objectives and Design Guidelines

- T1. Check that *corresponding simple names or expanded names* must be declared by the same declaration when they appear in conforming:
 - discriminant parts as default expressions or type marks (see IG 3.8.1/T3 and IG 7.4.1/T4);
 - formal parts as default expressions or type marks;
Implementation Guideline: Check for subprograms and generic subprograms (see IG 6.3/T9 and IG 10.2/T11).
Implementation Guideline: Check entry declarations and accept statements (see IG 9.5/T20).
 - deferred constant declarations as type marks (see IG 7.4.3/T3);
 - *type conversions of actual parameters* as type marks (see IG 6.4.1/T10).
- T2. Check that if different forms of a name are used in a default expression of:
 - a discriminant part (for private and incomplete types);
 - a formal part (for subprogram specifications);
 then the selector may not be an operator symbol or a character literal (RM 6.3.1/3).
- T3. Check that *universal_real* literals in default expressions must have the same value in conforming formal parts.

6.3.2 Inline Expansion of Subprograms

Semantic Ramifications

- S1. An implementation is free to expand any subprogram call inline, since there is no semantic effect of doing so. The absence of the pragma does not ensure against inline expansion, and the use of the pragma does not guarantee inline expansion. The pragma is of use only if a compiler is prepared to perform inline expansion and has not chosen to do so.
- S2. If a pragma does not appear in the required place, or its arguments do not correspond to what the RM requires, then the pragma must be ignored (RM 2.8/9). Hence, if any INLINE argument is not a subprogram or a generic subprogram name, then the entire pragma must be ignored, in principle. However, since there is no semantic effect associated with inline expansion, there is no way to tell whether the pragma has been ignored. In addition, since an implementation is always allowed to expand a call inline, it is certainly free to do so for the recognizable subprogram names in such a pragma.
- S3. Note that for each call of a subprogram named in the INLINE pragma, the implementation is free to follow or to ignore the recommendation.
- S4. INLINE pragmas only have an effect if they appear in the same declarative part as does the subprogram specification to which the pragma applies, unless the pragma applies to a library

unit. In that case, the pragma must appear following the library unit and preceding the subsequent compilation unit, e.g.:

```

procedure P is
    ...
end P;
pragma INLINE (P);
procedure Q is ... end Q;

```

S5. Note that INLINE can be specified before or after the body of its named subprogram has been declared.

S6. Note that the INLINE pragma may appear in the private part of a package specification and name a subprogram that appears in the visible part.

Changes from July 1982

S7. There are no significant changes.

Changes from July 1980

S8. The pragma INLINE is allowed for library units.

S9. The effect of the pragma INLINE on generic units is now specified.

Legality Rules

L1. An INLINE pragma must be ignored if:

- it does not appear immediately after a declarative item or immediately after a library unit;
- it appears in a package specification and one subprogram or generic subprogram named in the pragma was not declared earlier in the same package specification (by a subprogram declaration, generic subprogram declaration, generic instantiation, or renaming declaration);
- it appears in a declarative part and one subprogram or generic subprogram named in the pragma was not declared earlier in the same declarative part;
- it appears after a library unit and there is more than one name in the pragma's argument list or the single name is not the name of the preceding library unit;
- all names in the pragma's argument list are not names of subprograms or generic subprograms (e.g., one name is a name of a subprogram declared as a generic formal parameter).

Test Objectives and Design Guidelines

T1. Check that legal INLINE pragmas are recognized. (A warning may be issued if the recommendation is going to be ignored.)

T2. Check that illegal INLINE pragmas have no effect on the program unit. (Warnings should be issued).

Implementation Guideline: Include the following illegal cases:

1. The pragma appears in a declarative region and
 - the name is not the name of a subprogram or a generic subprogram;
 - the name is not the name of a subprogram or generic subprogram declared in the same declarative part or package specification.

Implementation Guideline: Use the name of a generic formal subprogram for one case

2. The pragma appears following a library unit and
 - the name is not the name of the library unit;
 - the library unit is not a subprogram or a generic subprogram.
3. The pragma does not appear in a declarative part, package specification, or following a library unit.

6.4 Subprogram Calls

Semantic Ramifications

S1. The semantic details of the association between actual and formal parameters are discussed in IG 6.4.1.

S2. Although programmers are not supposed to write programs so the order of actual parameter evaluation changes the effect of a call, a program that *is* affected by the order of parameter evaluation is not erroneous; it is simply "incorrect" (RM 1.6/9). Since it is not erroneous, an implementation cannot optimize programs based on the assumption that what is true for one order of evaluation is true for all orders of evaluation:

```

I : INTEGER;
S : STRING(1..5);

function CHANGE_I return INTEGER is
begin
    I := -1;
    return 3;
end;
...
I := 5;
P (CHANGE_I, S(I));

```

An implementation is free to evaluate S(I) first without checking at run-time to see if the value of I is acceptable, since constant propagation at compile-time allows the check to be omitted. If CHANGE_I is evaluated first, however, the evaluation of S(I) must raise CONSTRAINT_ERROR. If the RM had said that dependence on the order of parameter evaluation makes a program erroneous, an implementation would have been free to assume that, for nonerroneous programs, all evaluation orders have the same semantic effect, and so it would have been valid to omit the subscript check when evaluating S(I) in the call to P. This example (due to Paul N. Hilfinger) shows how incorrect order dependence and erroneousness differ.

Changes from July 1982

S3. There are no significant changes.

Changes from July 1980

S4. A function call with no actual parameters no longer requires an empty parameter list, ().

Legality Rules

L1. For a function or procedure call with positional parameters only:

- the number of actual parameters must equal the number of formal parameters (RM 6.4/5); or
- the number of actual parameters must be less than the number of formal

parameters, and, if N parameters are omitted, the last N formal parameters must have default values specified for them (RM 6.4/5);

- the base type of the *i*th actual and formal parameter must be the same (RM 6.4.1/1).

L2. For a function or procedure call with both named and positional parameters:

- the total number of actual parameters must not exceed the number of formal parameters (RM 6.4/5);
- omitted actual parameters must correspond to formal parameters for which default values were specified (RM 6.4/5);
- positional parameters must appear first (RM 6.4/4);
- a named parameter must not be specified for a formal parameter if an actual positional parameter is also given for that formal parameter (RM 6.4/5);
- the base types of corresponding formal and actual parameters must be the same (RM 6.4.1/1).

L3. The formal parameter name in a named parameter association must be identical to that of a formal parameter in the corresponding subprogram specification (RM 6.4/3).

L4. No duplicates are permitted among the formal parameter names used in an actual-parameter-part (RM 6.4/5).

Exception Conditions

See IG 6.4.1/E and 6.4.2/E.

Test Objectives and Design Guidelines

T1. Check that the form *F()* is illegal.

T2. For functions and procedures having no default parameter values, check that the number of actual positional and named parameters must equal the number of formal parameters.

Implementation Guideline: Use calls that are valid except for the number of parameters. Check calls to procedures and functions. Use purely positional notation, purely named notation, and a combination of positional and named notation.

Check that parameterless subprograms can be called with the appropriate notation.

Check that the base type of formal and actual parameters must be the same.

Implementation Guideline: Check numeric types in particular.

T3. Check that for a mixture of named and positional notation, named parameters cannot be interleaved with, and cannot precede, positional parameters.

Implementation Guideline: Use a call in which the types of all the formal parameters are identical. Check that in a mixture of named and positional notation, a named parameter and a later positional parameter cannot be specified for the same formal parameter.

Check that two or more named parameters cannot specify the same formal parameter.

Check that the name used in a named parameter must only be a name of a formal parameter.

Check that a formal parameter in a named parameter association is not confused with an actual parameter identifier having the same spelling (see IG 8.3.e/T3).

Check that a named parameter cannot be provided for a formal parameter if a positional parameter has already been given for that formal parameter.

T4. For functions and procedures having at least one default parameter, check that:

- calls of the form *F(A.,B)* are forbidden, where the second formal parameter has a default value;

- for a call using only positional notation, no parameters can be omitted unless the default parameters are at the end of the parameter list;
- for a call using named notation, omitted parameters must have default values;
- for a call using named notation, regardless of the order of the actual parameters, the correct correspondence with the intended formal parameter is achieved.

- T5. Check that a subprogram can be called recursively, and that global variables, local variables, and parameters of enclosing, recursively called subprograms are properly accessed from within recursive invocations.
- T6. Check that calls of the form $F(A|B \Rightarrow 0)$ are not permitted even if A and B are both integer formal parameters of F.

6.4.1 Parameter Associations

Semantic Ramifications

S1. The various syntactic forms allowed for names are given in RM 4.1/2. Of these forms, the term *variable_name* includes just *simple_name*, *indexed_component*, *selected_component*, and *slice*. The term excludes character literals, operator symbols, and attributes since these forms of name do not denote variables. In addition, the name of a constant is disallowed (e.g., the name of a constant object, loop parameter, subprogram in formal parameter, or generic in formal parameter).

S2. For use as the prefix in the name of a variable, a *function_call* must return an access value (see IG 5.2/S). Furthermore, since operators can be overloaded to deliver access values, should "+" be overloaded to yield an access value, "+"(A,B).all can be used as a variable name (since the prefix has the form of a function call).

S3. Numeric conversions raise *NUMERIC_ERROR* instead of *CONSTRAINT_ERROR* if the value being converted cannot be represented correctly (RM 3.5.4/10), i.e., if the value lies outside the range of the base type. Hence, the use of a type conversion as an in out or out parameter can raise *NUMERIC_ERROR*. For example:

```
procedure F (X : in out LONG_INTEGER);
  INT : INTEGER range 0..5;
  .
  F (LONG_INTEGER(INT));
```

The conversion to the formal type, *LONG_INTEGER*, cannot raise any exception. However, the inverse conversion from *LONG_INTEGER* to *INTEGER* will raise *NUMERIC_ERROR* if the value of X is not in the set of *INTEGER* values. *CONSTRAINT_ERROR* will be raised if the value is in the range of *INTEGER* values but not in the range 0..5.

S4. The semantic effect of processing an actual parameter depends on the mode, type, and form of the parameter. For parameters with mode in:

- the actual parameter is an expression and is evaluated (RM 6.4.1/2);
- if no exception is raised when the expression is evaluated, the value is checked against the subtype of the formal parameter (RM 6.4.1/6);
- if no exception is raised by the subtype check, the value of the actual parameter becomes the value of the formal parameter; in particular,

- for scalar and access types, and for private types whose full declarations declare a scalar or access type (RM 6.2/8), the value is copied (RM 6.2/6);
- for record, array, and task types, the value is copied or passed by reference (RM 6.2/7).

S5. For parameters with mode in out:

- before the call:
 - the variable name is evaluated (see IG 4.1.1, IG 4.1.2, and IG 4.1.3 for a discussion of what exceptions can be raised; note also that any function call appearing in the prefix of the variable name is evaluated (RM 4.1/10));
 - if no exception is raised when the name is evaluated, and if the actual parameter has the form of a type conversion, the conversion is performed (RM 6.4.1/4);
 - the (possibly converted) value of the actual parameter is checked against the subtype of the formal parameter (RM 6.4.1/6). Note that this check is also performed as part of type conversion (RM 4.6/4), and so is redundant if the actual parameter has the form of a type conversion;
 - if no exception is raised by the subtype check, the (possibly converted) value of the actual parameter becomes the initial value of the formal parameter, just as for in parameters (RM 6.2/6, /7);
- after the subprogram returns normally:
 - if the actual parameter has the form of a type conversion, the inverse conversion to the subtype of the actual parameter is performed (RM 6.4.1/4);
 - if the parameter has a scalar or access type, the (possibly converted) value of the formal parameter is checked against the subtype of the actual parameter (RM 6.4.1/7);
 - if no exception is raised by this subtype check, the value of the formal parameter becomes the value of the actual parameter.

S6. For parameters with mode out:

- before the call, the variable name is evaluated (as for an in out parameter);
- subsequent actions depend on the type of the parameter and on whether the actual parameter specifies a type conversion:

	scalar	access	array	record
perform type conversion?	no	yes*	yes*	yes*
		*see discussion below		
can exception be raised independent of subtype check?	no	yes	yes	no
check against formal subtype?	no	no	yes	yes

pass value of actual parameter?	no	yes	bounds	discrim
• after the call:				
for a type conversion, can an exception be raised?	yes	yes	no	no
check (converted) value against subtype of actual?	yes	yes	no	no

S7. If an actual out parameter has the form of a type conversion, the type conversion is performed if the parameter is *not* a scalar type even though RM 6.4.1/4 does not explicitly mention performing type conversions for out parameters, and superficially, by its explicit reference to type conversions for in out parameters, appears to exclude the evaluation of out parameter type conversions before the call. However, consideration of other requirements of the RM leads to the conclusion that the actual parameter type conversions must be evaluated before the call for nonscalar parameters. In particular, for access types, RM 6.2/6 requires that the value of the actual parameter be copied to the formal parameter. If the actual parameter has the form of a type conversion, such copying cannot be performed unless the conversion is done first since, without the conversion, there is no value to be copied. (From an implementation viewpoint, the access types might have different representations, e.g., one might be an offset pointer and the other an absolute address, so a physical conversion would be needed in any event.) Performing the type conversion requires checking the subtype of the value (RM 4.6/4). For access types, this means checking that any constraint imposed on the subtype of the formal parameter is satisfied by the actual variable's current value. For example,

```

type AST is access STRING;
subtype AST_3 is AST(1..3);
subtype AST_5 is AST(1..5);

X_3 : AST_3 := new STRING(1..3);

procedure P(X : out AST_5);
... P(AST_5(X_3)); ...

```

The call to P raises CONSTRAINT_ERROR prior to the call since the conversion to AST_5 raises CONSTRAINT_ERROR (RM 4.6/4 and RM 3.8/6). Note that without the conversion, no exception would be raised before the call:

```
P(X_3);      -- no exception before call
```

No exception is raised since RM 6.4.1/6 does not specify that the value of the actual parameter be checked against the subtype of the formal parameter. In the above case, the value of X_3 will be copied to the formal parameter (RM 6.2/6), even though it does not satisfy the formal parameter's constraint. This causes no inconsistencies since the formal parameter's value cannot be read within P (RM 6.2/5) nor can the bounds or subcomponents of the designated object be read (RM 4.1/4). In addition, no assignments can be made to the designated object or to its subcomponents, because such assignments would require component selection, which is forbidden by RM 4.1/4. If no value is assigned to X within P, then upon P's return, the original access value is copied back to X_3, and no exception is raised, since the designated object satisfies X_3's constraint. If a value is assigned to P.X, then the designated object must obey P.X's constraint (RM 6.2/9, RM 5.2/3, and RM 3.8/6). When P returns, CONSTRAINT_ERROR will be raised when it is found that the object designated by the formal parameter does not satisfy X_3's constraint (RM 6.4.1/7, /10).

s8. If the formal parameter has an unconstrained array type, then it is necessary to evaluate the type conversion since the index base types of the formal and actual parameters can be different (RM 4.6/11) and it is required to make the bounds of the actual parameter accessible to the formal parameter (RM 6.2/5). By requiring that the bounds be made accessible, the RM implies that the type conversion must be evaluated, since bounds of an incorrect type cannot be made available to the formal parameter. In addition to converting index bounds, an array conversion checks to ensure that any constraints on the array component type are identical. Finally, if the conversion is to a constrained array type, corresponding dimensions of the variable and the formal parameter must have the same number of components for non-null arrays; for null arrays, corresponding dimensions can have different values for the 'LENGTH attribute (RM 4.6/11 and RM 4.5.2/5).

s9. There are four ways of checking that an array type conversion for out parameters is performed before the call:

- if the formal and actual array component constraints are not the same, the array type conversion will raise CONSTRAINT_ERROR if it is performed:

```

type AR_3 is array (INTEGER range <>) of STRING(1..3);
type AR_5 is array (INTEGER range <>) of STRING(1..5);
subtype AR_5C is AR_5(1..2);

Y_3 : AR_3(1..2);

procedure Q (X : out AR_5);      -- unconstrained array formal
procedure R (X : out AR_5C);     -- constrained array formal
...
Q (AR_5(Y_3));                  -- conversion to unconstrained array type
R (AR_5C(Y_3));                  -- conversion to constrained array type

```

An evaluation of the conversions in the calls to Q and R raises CONSTRAINT_ERROR since the component constraint of Y_3 is not identical to the component constraint for AR_5 and AR_5C. Hence, the failure to evaluate the conversion can be detected.

- if the actual parameter is converted to a constrained array type, the bounds of the formal parameter are used and the bounds of the actual variable are not passed to the formal parameter or checked against the formal parameter bounds (the type conversion does check that 'LENGTH is the same for corresponding formal and actual parameter dimensions when the formal specifies a non-null array):

```

subtype ST3 is STRING(1..3);
subtype ST5 is STRING(3..5);

XS_3 : ST3;

procedure S (X : out ST5);
...
S(ST5(XS_3));                  -- no exception
S(XS_3);                       -- CONSTRAINT_ERROR before call

```

The first call to S raises no exception because the evaluation of the conversion to ST5 implies that the bounds of the actual parameter (i.e., the bounds of the type conversion) are 3..5, and these bounds satisfy the formal parameter's

constraint. The second call raises `CONSTRAINT_ERROR` since the bounds of `XS_3` are not the same as the bounds specified for the formal parameter. Hence, if the conversion is not performed, the first call will incorrectly raise an exception.

- if an actual parameter is converted to a constrained array type that is a null array, the semantics for type conversion only require that `CONSTRAINT_ERROR` be raised if the array variable is not null ("a check is made that for each component of the operand there is a matching component of the target subtype"; RM 4.6/13 and RM 4.5.2/5). Corresponding dimensions need not have the same number of components since if both arrays are null, they clearly have the same number of matching components, namely, none. When an actual out parameter does not have the form of a type conversion, the bounds of the variable must satisfy the formal's constraint, i.e., corresponding dimensions must have the same bounds (RM 3.6.1/4). There is no special rule for null arrays in this case. Hence, consider the following example:

```

type ARR is array (INTEGER range <>,
                  INTEGER range <>) of INTEGER;
X : ARR(1..0, 1..2);
subtype NARR is ARR(1..0, 1..3);
procedure P (X : out NARR);
...
P(X);                -- CONSTRAINT_ERROR
P(NARR(X));          -- no exception

```

The first call raises `CONSTRAINT_ERROR` because the bounds of the formal and actual are not identical. For the second call, the type conversion does not raise an exception, because `X` is a null array and `NARR` requires a null array. The bounds passed to the formal are the bounds specified for `NARR` (not the bounds of `X` (RM 4.6/11)), so the bounds of the converted actual parameter match the bounds of the formal parameter.

- if an actual parameter is converted to an unconstrained array type, the bounds of the variable must lie within the range of each index subtype (RM 4.6/13):

```

NULL_STR : STRING(1..0) := "";
procedure U (X : out STRING);
...
U(NULL_STR);                -- no exception
U(STRING(NULL_STR));        -- CONSTRAINT_ERROR

```

No exception is raised for the first call since the null string belongs to the `STRING` type. `CONSTRAINT_ERROR` is raised by the evaluation of the type conversion in the second call since `NULL_STR'LAST` does not belong to `STRING`'s index subtype, `POSITIVE`.

Similarly, if the bounds do not even lie within the range of the index base type, `NUMERIC_ERROR` will be raised by a type conversion:

```

type LONG is array (LONG_INTEGER range <>) of INTEGER;
type SHORT is array (INTEGER range <>) of INTEGER;
LONG_ARR : LONG (LONG_INTEGER(INTEGER'LAST) + 1 .. 0);
procedure V (X : SHORT);
...
V (SHORT(LONG_ARR));        -- NUMERIC_ERROR is raised

```

NUMERIC_ERROR is raised since INTEGER'LAST + 1 lies outside the range of SHORT's index base type, INTEGER.

S10. For an out parameter having a record type without discriminants, a type conversion does not affect the type of any subcomponent. In addition, the only information that must be passed to the formal parameter before the call is the discriminants of the record (if any) and the bounds, discriminants, access values, and task type values associated with subcomponents of the actual parameter. All this information can be extracted and passed to the formal parameter without converting the actual variable. Hence, the RM does not imply that a type conversion used as an actual out parameter is evaluated before the call. However, there is no way to check whether such a conversion is evaluated or not, since the conversion cannot raise an exception except when the record type has discriminants, and in this case, an exception would be raised by the conversion if and only if an exception would be raised when the value of the discriminants is checked against formal parameter's discriminants. Hence, there is no way to determine whether the conversion is performed for record types. For simplicity and by analogy with array types, it is acceptable to assume that the conversion is performed.

S11. Use of a type conversion as an out parameter for an array or record type still allows the out parameter to be passed by reference if the physical representation of the formal and actual types is the same, although some care needs to be taken with the bounds of constrained formal array parameters to ensure that the bounds of the formal parameter are used instead of the bounds of the actual parameter.

S12. When an actual in out or out parameter has the form of a conversion to an unconstrained record type, then the formal parameter is also an unconstrained record type (since the type mark in the conversion and in the formal parameter must be the same (RM 6.4.1/3)) and the value of 'CONSTRAINED' for the formal parameter is determined by the value of 'CONSTRAINED' for the actual parameter's variable (RM 6.2/9). Even though the conversion is performed for in out parameters (RM 6.4.1/4), 'CONSTRAINED' is not necessarily TRUE for the formal parameter, i.e., it would be a mistake to treat the type conversion as an expression yielding a value, for which 'CONSTRAINED' is TRUE.

S13. For a scalar out parameter having the form of a type conversion, the type conversion is not performed because the value is not passed to the formal parameter and the RM does not require the evaluation of the out parameter before the call. Hence, if the value of the actual variable does not satisfy the formal's constraint, no exception is raised:

```
subtype SMALL is INTEGER range 0..10;
procedure U (X : out SMALL);
X_INT : INTEGER := 15;
...
U (SMALL(X_INT)); -- no exception before call
```

S14. When a type conversion is used as an expression, its argument may be enclosed in parentheses (RM 4.6/3), e.g., if STRING(X) is legal, so is STRING((X)). However, the syntactic form of actual in out and out parameters is fully specified in RM 6.4.1/3, and no extra parentheses are allowed.

S15. A subprogram call is erroneous if a scalar actual in or in out parameter has an undefined value when the subprogram is called since such parameter values must be evaluated (RM 3.2.1/18). Similarly, a call is erroneous if a scalar out formal parameter has an undefined value on completion of the subprogram body. In all other cases, the values read as part of the semantics of a subprogram call are defined:

- an access type object always has a defined value.

- only the bounds of an array object are read (i.e., checked) and the bounds are always defined.
- only discriminant values are read (i.e., checked) for objects that have discriminants, and these values are always defined.

In the absence of this rule concerning erroneous calls, a subprogram would have to check the value of scalar parameters against formal parameter constraints even when it appeared that such a check could be omitted. As it is, a compiler is allowed to omit range checks for scalar parameters under the following circumstances:

- when calling a subprogram, for in and in out parameters, if the subtype range of the actual scalar parameter is contained within the subtype range of the formal parameter:

```

subtype SMALL is INTEGER range 0..10;
subtype LARGE is INTEGER range -10..10;
procedure P (X : in LARGE; Y : in out LARGE);
V : SMALL;
...
P(V, V);           -- no constraint check needed before call

```

- when returning from a subprogram, if the subtype of a formal in out or out parameter is contained within the subtype range of the actual parameter.

These checks can be omitted because, if the parameter has a defined value, the checks are unnecessary, and if the parameter does not have a defined value, the effect of the program is undefined (RM 1.6/7).

S16. Out parameters behave in many ways as if they were targets of an assignment statement. However, CONSTRAINT_ERROR can be raised by a subprogram call even when assignment of a formal parameter value to an actual variable would not raise CONSTRAINT_ERROR:

```

type T (D : INTEGER := 0) is
  record null; end record;
subtype T5 is T(5);

A : T := (D => 3);           -- no exception raised

procedure P (X : out T5) is
begin
  X := (D => 5);
end;
...
P(A);                       -- CONSTRAINT_ERROR raised
A := (D => 5);               -- no exception raised

```

CONSTRAINT_ERROR is raised by P(A) since the value of A.D (i.e., 3) does not equal the value of P.X.D (i.e., 5), and these values are checked before the call.

S17. The type mark used in an in out or out actual parameter type conversion must conform to the name used in the formal parameter declaration. Conformance means that the names must denote the same declaration (IG 6.3.1/S). In particular, a name declared by a subtype declaration does not conform to a name declared by a type declaration, and vice versa. Note that conformance is not required for type conversions used as in actual parameters:

```

subtype SMALL is INTEGER range 1..5;
subtype SMALLISH is SMALL;

F : FLOAT := 1.0;
procedure P (X : SMALL;
             Y : in out SMALL);
...
P (SMALLISH(1.0),    -- legal; base types are the same
  SMALLISH(F));      -- illegal; SMALL and SMALLISH do not conform

```

SMALLISH(F) is legal for the first parameter because the first parameter is an expression corresponding to an in parameter. The only requirement here is that the base type of the formal and actual parameter be the same. The second parameter is, however, illegal since the type mark in the conversion does not denote the same declaration as the type mark used in the in out formal parameter's declaration.

S18. Note that the RM does not define the order in which actual parameters are updated when returning from a call (RM 6.4/6). In particular, it is possible for some actual parameters to have their values updated before a constraint check fails for a parameter and an exception is raised.

S19. When a formal parameter has a private type, the constraint checks required are those required for the full declaration of the private type, although RM 6.4.1/9 may seem, at first, to imply otherwise. However, RM 6.2/8 implies that the parameter passing semantics of private types is determined by the rules that apply to the private type's full declaration, including the rules for constraint checking. The following example (conceived by David A. Taffs) shows why the value of a formal out (or in out) parameter of a private type must sometimes be checked against the actual parameter's constraints when a subprogram returns, even when the formal parameter is a private type:

```

package P is
  type T is private;
  DC : constant T;

  generic
    package INVOKE_Q is          -- instantiation invokes Q
    end P;
  private
    type T is new INTEGER;
    DC : constant T := -1;
  end P;

  procedure Q (X : out T) is
  begin
    X := P.DC;
  end Q;

  generic
    Y : in out P.T;
  package CALL is
  end CALL;

```

```

package body CALL is
begin
    Q (Y);      -- note call occurs outside P; Y has a private type
end CALL;

package body P is
    Z : T range 0..1 := 0;
    package body INVOKE_Q is
        package CALL_Q is new CALL (Z);
    end INVOKE_Q;
end P;

package CALL_Q_NOW is new P.INVOKE_Q;

```

The elaboration of CALL_Q_NOW will execute the call to Q that occurs in the body of CALL. This call will assign the value of P.DC to Q's formal parameter, and when Q returns, an attempt will be made to assign the value of P.DC to P.Z. This attempt will raise **CONSTRAINT ERROR** since the value of the formal parameter must be checked against the actual parameter's constraint.

Changes from July 1982

S20. An actual parameter associated with a formal parameter of mode in out must not be a subcomponent of, or a formal parameter of, mode out.

S21. The execution of a program is erroneous if the value of an actual in or in out scalar parameter is undefined when the subprogram is invoked, or if the value of a formal out scalar parameter is undefined when the subprogram returns.

Changes from July 1980

parameter.

S22. It is now stated explicitly that actual in out parameters having the form of a type conversion are converted to the specified type before the call, and formal in out or out parameters are converted to the type of the actual parameter after (normal) completion of the subprogram.

S23. The type mark specified for actual parameters having the form of type conversions and the type mark used in the formal parameter declaration must conform in accordance with the rules of RM 6.3.1.

Legality Rules

- L1. An actual out or in out parameter must be either the name of a variable or have the form of a type conversion applied to the name of a variable (RM 6.4.1/3).
- L2. The base types of the formal and actual parameters must be the same (RM 6.4.1/1).
- L3. The type mark appearing in an actual in out or out parameter having the form of a type conversion must conform to the type mark of the formal parameter (RM 6.4.1/3).
- L4. An actual in out variable must not be a formal parameter of mode out or a subcomponent of such a formal parameter (RM 6.4.1/3).
- L5. An actual in parameter must not be a formal parameter of mode out or a subcomponent of such a formal parameter, unless the subcomponent is a discriminant (RM 6.2/5).

Exception Conditions

E1. For an actual parameter of mode *In*, **CONSTRAINT_ERROR** is raised if:

- the formal parameter has a scalar type and the value of the actual parameter before the call lies outside the range specified for the formal parameter.
- the formal parameter has a constrained array type and the index bounds of the actual parameter are not equal to the bounds specified for the formal parameter.
- the formal parameter has a constrained record, private, or limited private type and the discriminant values for the actual parameter do not equal those specified for the formal parameter.
- the formal parameter has a constrained access type, the value of the actual parameter is not null, and the bounds or discriminants of the designated object do not equal the values of the bounds or discriminants specified for the formal parameter's constraint.

E2. For a parameter of mode *In out* having the form of a variable name:

- **CONSTRAINT_ERROR** is raised before the call if:
 - the parameter has a scalar type and the value of the variable lies outside the range specified for the formal parameter.
 - the formal parameter has a constrained array type and the index bounds of the actual parameter are not equal to the bounds specified for the formal parameter.
 - the formal parameter has a constrained record, private, or limited private type and the discriminant values for the variable do not equal those specified for the formal parameter.
 - the formal parameter is a constrained access type, the value of the variable is not null, and the bounds or discriminants of the designated object do not equal the values of the bounds or discriminants specified for the formal parameter's constraint.
- **CONSTRAINT_ERROR** is raised after normal completion of the subprogram if:
 - the formal parameter has a scalar type or a private type whose full declaration declares a scalar type and the value of the formal parameter lies outside the range specified for the variable named as the actual parameter.
 - the actual parameter has a constrained access type or a private type whose full declaration declares a constrained access type; the formal parameter's value is not null; and the bounds or discriminants of the formal parameter's designated object do not equal the values of the bounds or discriminants specified for the actual variable's subtype.

E3. For a parameter of mode *In out* having the form of a type conversion applied to the name of a variable:

- **NUMERIC_ERROR** is raised before the call if:

- the parameter has a scalar numeric type and the value of the actual parameter cannot be accurately represented as a value of the formal parameter's type because the value lies outside the range of the formal parameter's base type (RM 3.5.4/10).
 - the formal parameter has an unconstrained array type, and for some dimension of the formal parameter's type, an index bound of the variable lies outside the range of the formal parameter's index base type (RM 4.6/13).
- **CONSTRAINT_ERROR** is raised before the call if:
 - the parameter has a scalar type and the converted value of the variable lies within the range of the formal parameter's base type but outside the range specified for the formal parameter.
 - the formal parameter has an array type:
 - constraints are specified for the component type of the variable and the component type of the formal parameter, and the constraints are not equal (RM 4.6/13); or
 - the array type is unconstrained, the operand is a non-null array, and for some dimension of the formal parameter's type, the index bounds of the variable, after conversion to the formal parameter's index base type, do not both lie within the range of the formal parameter's index subtype (RM 4.6/13).
 - the array type is constrained,
 - the formal parameter declares a null array, and the value of the variable is not a null array (RM 4.6/13); or
 - the formal parameter does not declare a null array, and for at least one dimension, the number of components specified for the variable of the variable is not the same as the number of components specified for the formal parameter (RM 4.6/13).
 - the formal parameter is a constrained access type, the value of the variable is not null, and the bounds or discriminants of the designated object do not equal the values of the bounds or discriminants specified for the formal parameter's constraint.
 - **NUMERIC_ERROR** is raised after completion of the subprogram if the parameter has a scalar numeric type and the value of the formal parameter cannot be accurately represented as a value of the actual parameter's type because the value lies outside the range of the actual parameter's base type (RM 3.5.4/10).
 - **CONSTRAINT_ERROR** is raised after completion of the subprogram if:
 - the parameter has a scalar type or a private type whose full declaration declares a scalar type and the converted value of the formal parameter lies within the range of the actual parameter's base type but outside the range specified for the actual variable.
 - the formal parameter is a constrained access type or a private type

whose full declaration declares a constrained access type, the value of the formal parameter is not null, and the bounds or discriminants of the designated object do not equal the values of the bounds or discriminants specified for the actual parameter.

E4. For a parameter of mode out having the form of a variable name:

- CONSTRAINT_ERROR is raised before the call as for an in out formal parameter of a constrained array, record, private, or limited private type (see E2).
- CONSTRAINT_ERROR is raised after normal completion of the subprogram as for in out parameters (see E2).

E5. For a parameter of mode out having the form of a type conversion applied to the name of a variable:

- NUMERIC_ERROR is raised before the call as for an in out formal parameter of an unconstrained array type (see E3).
- CONSTRAINT_ERROR is raised before the call as for in out formal parameters having an array or constrained access type (see E3).
- NUMERIC_ERROR is raised after completion of the subprogram as for an in out formal parameter of a scalar type (see E3).
- CONSTRAINT_ERROR is raised after completion of the subprogram as for an in out formal parameter of a scalar, private, or constrained access type (see E3).

Test Objectives and Design Guidelines

T1. Check that the expression corresponding to an out or in out parameter cannot be:

- a constant, including an in formal parameter, an enumeration literal, a loop parameter, a record discriminant, the literal null, and a named number;
- a parenthesized variable;
- a type conversion with a parenthesized variable;
- a function returning a value of a record, array, private, scalar, or access type;
- an attribute;
- an aggregate, even one consisting only of variables;
- a qualified expression containing only a variable name;
- an allocator;
- an expression containing an operator.

T2. Check that the base types of corresponding formal and actual parameters must not be different (see IG 6.4/T2).

T3. For type conversions of a scalar variable as an in out parameter, check that:

- NUMERIC_ERROR is raised for numeric types:
 - before the call when the actual value lies outside the range of the formal parameter's base type.

- after the call when the formal parameter's value lies outside the range of the actual variable's base type.
- **CONSTRAINT_ERROR** is raised:
 - before the call when the converted value of the actual variable lies outside the range of the formal parameter's subtype.
 - after the call when the converted value of the formal parameter lies outside the range of the actual variable's subtype.

For a type conversion of an array variable as an in out or out parameter, check that:

- **CONSTRAINT_ERROR** is raised before the call if:
 - the subtype constraints imposed on the actual variable's components are not the same as the constraints imposed on the formal parameter's components;
Implementation Guideline: Check conversion to both a constrained and an unconstrained array type.
 - for conversion of a non-null value to an unconstrained array type, an index bound of the actual parameter, after conversion, does not lie within the range of an index subtype of the formal parameter.
Implementation Guideline: For a null multidimensional actual array, **CONSTRAINT_ERROR** is raised if a non-null bound is outside the index subtype (AI-00313).
 - for conversion to a constrained array type, the number of components per dimension is not the same for the formal and actual parameters when the actual variable is a non-null array, or the formal parameter specifies a non-null array.
- **NUMERIC_ERROR** is raised before the call for conversion to an unconstrained array type if the value of a bound of the variable lies outside the range of the corresponding index base type.

For a type conversion to a constrained access type as an in out or out parameter, check that **CONSTRAINT_ERROR** is raised:

- before the call if the value of the actual parameter is not null and the bounds or discriminants of the designated object do not equal the bounds/discriminants of the formal parameter.
Implementation Guideline: Check for both null and non-null array objects.
- after the call, if the value of the formal parameter is not null and the bounds or discriminants of the designated object do not equal the bounds/discriminants of the actual variable.

T4. For calls not involving parameters having the form of a type conversion, check that **CONSTRAINT_ERROR** is raised under the appropriate circumstances, namely:

- before the call, when the value of a scalar in or in out actual parameter does not satisfy the range constraint of the formal parameter;
- after the call, when the value of a formal out or in out scalar parameter does not satisfy the range constraint of the actual parameter at the time of normal subprogram return;

- before the call, for all modes, when an actual record parameter has discriminant values that are not equal to the discriminant values of the formal parameter;

Implementation Guideline: In particular, try an unconstrained actual out parameter.

- before the call, for all modes, when an actual array parameter has different bounds for one dimension than is required by the constrained formal parameter;

Implementation Guideline: Check for null arrays with index values that are outside the index subtype.

Implementation Guideline: Check that null multi-dimensional actual parameters must have the same bounds as the formal parameter.

- for access types, when the index bounds of the object designated by the actual variable do not equal the index bounds specified for the formal parameter:

- before the call for in and in out parameters;
- after the call for in out and out parameters.

- for access types, when the discriminant values of the object designated by an actual variable do not equal the discriminant values specified for the formal parameter:

- before the call for in and in out parameters;
- after the call for in out and out parameters.

Implementation Guideline: Check that within a procedure, assignments to an out parameter obey the constraints of the formal parameter, not the constraints of the actual variable, when the constraints of the formal and actual parameter are different.

- after the call, when the discriminant values or index bounds associated with the value of an unconstrained formal access out or in out parameter do not equal the constraint values of a constrained actual access parameter.

Check that when the full declaration of a private type declares an access or scalar type and the private type is used as an out or in out parameter, `CONSTRAINT_ERROR` is raised after the call if the value of the formal parameter does not belong to the scalar or access subtype of the actual parameter.

Check that `CONSTRAINT_ERROR` is raised at the place of the call (i.e., within the caller, not within the called subprogram) in the above circumstances.

- T5. Check that `CONSTRAINT_ERROR` is not raised under the appropriate circumstances. In particular, check that no exception is raised:

- at the time of call, for all modes, when the value of a scalar actual out parameter does not satisfy the range constraints of the formal parameter;

Implementation Guideline: Check when the actual has the form of a type conversion as well as the form of a variable name.

- at the time of call, for all modes, when an actual access parameter has the value null and the formal parameter is constrained (even if the subtype of the actual parameter does not match that of the formal parameter), when the actual parameter has the form of a variable name or type conversion;

- on normal return, for in out and out parameters, when the formal parameter value is null and the actual parameter is constrained (even if the subtypes of the formal and actual parameters are not the same);

- for an in out or out parameter of an array type when the formal parameter is constrained and the actual parameter has the form of a type conversion:

- corresponding dimensions of the formal and actual parameter have the same number of components (for a non-null array), and the index bounds of the actual parameter lie outside the index subtype of the formal parameter.
- corresponding dimensions of the formal and actual parameter do not have the same number of components, but the formal and actual parameter are both null arrays.

- T6. Check that unconstrained record, private, limited private, and array formal parameters use the constraints of the actual parameter (even when the default parameter value has a constraint different from that of the actual parameter).

Implementation Guideline: For record, private, and limited private types having default discriminant constraints, be sure to try an uninitialized constrained variable as an out actual parameter.

Implementation Guideline: Check that null strings can have negative bounds, and the bounds are passed correctly.

Check that assignments to (formal parameters of) unconstrained record types without default constraints (i.e., for which `TCONSTRAINED` is always true) raise `CONSTRAINT_ERROR` if an attempt is made to change the constraint of the actual parameter (by making a whole-record assignment to the formal parameter).

Check that assignments to (formal parameters of) unconstrained record types with default constraints (i.e., for which `TCONSTRAINED` is true or false depending on its value for the actual parameter) raises `CONSTRAINT_ERROR` if the actual parameter is constrained and the constraint values of the object being assigned do not satisfy those of the actual parameter. Check that `CONSTRAINT_ERROR` is *not* raised if the actual parameter is unconstrained, even if the assignment changes the constraints of the actual parameter.

Implementation Guideline: Try a case where an actual parameter has the form of a type conversion.

Implementation Guideline: For both checks, include cases where the unconstrained record type is the full declaration of a private or a limited private type.

Implementation Guideline: Try these checks for nested procedure calls as well, i.e., where an unconstrained formal parameter is used as an actual parameter in a subprogram call.

- T7. Check that actual parameters are evaluated and identified at the time of call, e.g., use a call of the form `P(I, A(I))` where the parameters of `P` are out parameters, or use a name with a function prefix.

- T8. Check that all permitted forms of variable parameters are permitted.

Implementation Guideline: Include a case when the variable is named by a dereferenced function, i.e., by `F.all`.

- T9. Check that slices and arrays that are components of records are passed correctly to subprograms.

Implementation Guideline: Use all parameter modes.

Implementation Guideline: Use multidimensional arrays.

Implementation Guideline: Use records with components whose bounds depend on a discriminant and have more than one array component.

Implementation Guideline: Use objects designated by access types.

Implementation Guideline: Be sure that arrays with different bounds can be passed to unconstrained formal parameters.

Implementation Guideline: Pass a formal as an actual parameter.

- T10. Check that type marks appearing in actual in out or out parameters must conform to the type mark given in the formal parameter's declaration.

Check that conformance is not required for in parameters.

6.4.2 Default Parameters

Semantic Ramifications

S1. An implementation is not allowed to reject subprogram specifications with default values not in the range of a formal parameter's subtype. For example,

```
procedure P (X : NATURAL := -1) ...
```

must be accepted by the compiler. (CONSTRAINT_ERROR will be raised if the default value is used.)

S2. In parameters with default expressions are rather similar to constant object declarations initialized with the default value, e.g., an initial value can be provided for an unconstrained array type. There is an important difference, however — the value of a default expression "is used as an implicit actual parameter" and is not "assigned" to a formal parameter. Hence, default expressions can be specified for limited types, but not for constants of limited types. In addition, if a formal parameter has a constrained array type, there is no "sliding" of the default value:

```
subtype STR3 is STRING(1..3);
procedure P (X : STR3 := (3..5 => 'A'));
```

The call to P will raise CONSTRAINT_ERROR if the default value is used. Similarly, an array aggregate with an **others** choice is legal if named associations are used and if the formal parameter is constrained:

```
subtype STR2_4 is STRING(2..4);
S : STR2_4 := (2 => 'A', others => 'C');      -- illegal: RM 4.3.2/6
procedure Q (X : STR2_4 := (2 => 'A', others => 'C'));
```

The aggregate is legal as an actual parameter, but not as the value in an assignment context, e.g., in an object declaration.

Changes from July 1982

S3. There are no significant changes.

Changes from July 1980

S4. For omitted parameter associations, the default expression is evaluated before the call (instead of when the subprogram declaration is elaborated).

Exception Conditions

E1 For omitted parameter associations, CONSTRAINT_ERROR is raised:

- for scalar parameters if the value of the default expression lies outside the range of the formal parameter.
- for formal parameters having a constrained array type if the bounds of the default expression do not equal the bounds specified for the formal parameter.
- for formal parameters having discriminants if the discriminants for the value of the default expression do not equal the discriminants specified for the formal parameter.
- for formal parameters of a constrained access type if the value of the default expression is not null and the bounds or discriminants of the designated object do not equal the bounds or discriminants specified for the formal parameter.

- for formal parameters of a private type if the value of the default expression would raise an exception when the appropriate rule for the private type's full declaration is used.

Test Objectives and Design Guidelines

- T1. Check that default values of all types (including limited types) can be passed to a formal parameter. (Nonlimited types are tested in IG 6.1/T8.)

Check that `CONSTRAINT_ERROR` is raised for a default expression, if appropriate (see IG 6.1/T8).

Check that an aggregate with an `others` choice can be used as a default value for a parameter with a constrained array subtype (see IG 4.3.2/T4).

- T2. Check that default expressions are evaluated each time they are needed.

6.5 Function Subprograms

Semantic Ramifications

S1. The exception `PROGRAM_ERROR` must be raised when no other exception is raised and the last simple statement executed within a function body is not a return statement or does not propagate an exception out of the function. `PROGRAM_ERROR` is raised at the function call, not within the function body, since the RM says `PROGRAM_ERROR` is raised "if a function is left ..."; "is left" implies that no further execution is possible within the body. Moreover, RM 6.3/6 says a subprogram returns to its caller "upon completion of its body." This statement applies to both functions and procedures, and means that a subprogram, even a function, returns if there are no more statements to be executed. In the case of a function, however, this return immediately raises `PROGRAM_ERROR` at the point of the call.

S2. Note that an implementation is not required to determine whether or not a return statement can be executed. Furthermore, it cannot reject a function whose return statement is unexecutable. A warning can be issued if a compiler discovers that `PROGRAM_ERROR` will be raised on some or all calls. The exception `PROGRAM_ERROR` must be raised when no other exception is raised and a return statement is not executed.

Changes from July 1982

S3. The rule for raising `PROGRAM_ERROR` does not apply when the function is left by propagating an exception.

Changes from July 1980

S4. The exception `PROGRAM_ERROR` is raised when a function body is left other than by a return statement or by the propagation of an exception.

Legality Rules

- L1. All parameters of a function subprogram must have the mode `in`.
- L2. A function body must contain a return statement specifying a return value. This return statement must not be internal to nested program units.

Exception Conditions

- E1. The exception `PROGRAM_ERROR` is raised at the point of the function call when the last simple statement executed within a function body is not a return statement or does not propagate an exception out of the function.

Test Objectives and Design Guidelines

- T1. Check that In out and out parameters cannot be specified for functions.

Check that a return statement with a value specified is required inside a function body.

Implementation Guideline: Include a case where the return statement is inside a nested function, but not given in the containing function.

- T2. Check that a function must contain at least one return statement (excluding return statements nested in inner function bodies) and is not illegal if its only return statements occur in unexecutable sections of code.
- T3. Check that if no return statement is executed, a function raises PROGRAM_ERROR at the point of the call rather than within the function body.
- T4. Check that a function can propagate an exception out of its body without raising PROGRAM_ERROR.

6.6 Overloading of Subprograms

Semantic Ramifications

S1. Note that for packages, the visible and the private part of a package specification, and the declarative part in a package body, are all part of the same declarative region for purposes of deciding when conflicting declarations are present (RM 8.1/2 and RM 8.3/17). Similarly, the declarations in a task specification and body are part of the same declarative region (RM 8.1/2). Hence, subprograms declared in a task body can overload entries declared in a task specification (RM 9.5/5):

```
task type T is
    entry E (A : INTEGER);           -- E1
and T;

task body T is
    procedure E (A : FLOAT) ...      -- E2; overloads entry E
    procedure E (A,B : INTEGER := 0); -- E3; overloads previous E's
begin
    E(1.0);                         -- calls procedure E2
    E(1);                           -- ambiguous; calls entry E1 or procedure E3
end T;
```

Calls using a name that is overloaded for an entry or subprogram are resolved using the rules in RM 8.7 and RM 6.6/3. Note that there is no way to call entry E unambiguously.

S2. Since parameter names do not affect the parameter profile of a procedure, an inner procedure can hide an outer procedure with the same profile, but with different formal parameter names:

```
procedure P (A : INTEGER) is
    procedure P (B : INTEGER) is
        begin ... end P;
begin
    P(A => 1);                       -- illegal; outer P hidden
end P;
```

S3. Note that neither the mode of a formal parameter nor the form of an actual parameter is used to resolve a subprogram call (RM 6.6/3):

```

procedure P (A : in out FLOAT;
             B : FLOAT := 1.0);
procedure P (C : FLOAT := 2.0);
...
P(2.0);           -- ambiguous

```

Since the mode is ignored when resolving a call, P(2.0) is ambiguous even though P(C => 2.0) is the only legal call when the first parameter in the call is a constant. Note that if P.A were of mode in, then the above call could be either P(A => 2.0) or P(C => 2.0); hence, P(2.0) is ambiguous when parameter modes are ignored. See IG 8.7/S for further discussion.

S4. Generic subprograms are not considered subprograms immediately within the declarative region containing their declaration (RM 12.1/5). Hence, a generic subprogram and a subprogram cannot be declared with the same name in the same declarative region:

```

generic
procedure P;

procedure P (X : INTEGER);           -- illegal

procedure P is
  procedure P (Y : INTEGER) is ... end P; -- legal
begin null; end P;

```

Within a generic subprogram, the generic unit is considered a subprogram (RM 12.1/5). Hence, the inner declaration of P overloads the generic P.

S5. An enumeration literal has a parameter and result type profile equivalent to a parameterless function returning a value of the literal's type. Hence, it is illegal to declare a function and an enumeration literal having the same profile in the same declarative region:

```

type ENUM is (A, B);
function A return ENUM; -- illegal redeclaration of A

```

S6. Note that the definition of "overloaded" in RM 6.6/2 says that two subprograms are overloaded even if their scopes do not overlap. In such a case, the visibility rules, when applied to the name of the subprogram, suffice to resolve the overloading.

S7. RM 6.6/3 specifies what information in a subprogram call can be used to resolve the call. RM 8.7 specifies additional rules concerning how the context of a subprogram call can be used to resolve overloaded calls. In particular, RM 8.7 explains how the result type of a function may be used in resolving a function call. See IG 8.7/S for further discussion.

Changes from July 1982

S8. There are no significant changes.

Changes from July 1980

S9. The overloading resolution rule for calls now explicitly says the number of parameters is used.

S10. The names of formal parameters no longer affect the parameter profile of a subprogram, nor is the presence or absence of default expressions considered.

Legality Rules

L1. Two subprograms or single entries having the same designator cannot be declared in the same declarative region if (RM 8.3/17):

- they are both procedures or single entries, or are both functions; and
 - the number, order, and base types of the parameters are the same; and
 - for functions, the result base types are the same.
- L2. A subprogram call is not allowed unless the name of the subprogram, the number of parameter associations, the types and order of the actual parameters, the names of the formal parameters (if named associations are used), and the result type (for functions) do not suffice to determine which subprogram is being called. (Note that rules in RM 8.7 restrict how the result type of a function call can be used in helping to resolve which function is called.)

Test Objectives and Design Guidelines

- T1. Check that subprogram redeclarations are forbidden. Use two subprograms, two entries, or an entry and a subprogram declared in the same declarative region that are identical except for one of the following differences:
- the parameters are named differently (differences in parameter names are ignored).
 - the subtypes of a parameter are different (differences in subtype names are ignored if the base types are the same).
 - the result subtypes of two functions are different (differences in subtype names are ignored if the base types are the same).
 - the parameter modes are different; also try reordering the parameters and changing their modes.
 - a default expression is present/absent (the presence or absence of a default expression does not affect the parameter profile).

Check that a function declaration equivalent to an explicit enumeration literal is not allowed.

Check that a subprogram cannot have the same identifier as a variable, type, subtype, constant, exception, task unit, number, array, package, or generic subprogram declared previously in the same declarative region.

Implementation Guideline: Use a renaming declaration, a subprogram declaration, a subprogram body, a generic unit, and a generic formal declaration.

- T2. Check that overloaded subprogram declarations are permitted in which there is a minimal difference between the declarations. In particular, use declarations that differ in only one of the following aspects:
- one is a function; the other is a procedure.
Implementation Guideline: Try parameterless subprograms as well as subprograms having at least one parameter.
 - one subprogram has one less parameter than the other (the omitted parameter may or may not have a default value).
 - the base type of one parameter is different.
 - one subprogram is declared in an outer declarative part, the other is declared in an inner part, and
 - the parameters are ordered differently;

- one subprogram has one less parameter than the other, and the omitted parameter has a default value;
- the result types of two function declarations are different.

Implementation Guideline: Each of the subprograms in the above tests must be called, if possible, to ensure that the correct subprogram is invoked.

6.7 Overloading of Operators

Semantic Ramifications

S1. The restrictions given in RM 6.7/4 may seem to prohibit declaring a user-defined equality operator for nonlimited types. But, in fact, it is possible to provide a user-defined equality operator for any type. For example, one might like to define a type for polar coordinates so that an expression like $(-90, 5) = (270, 5)$ would be TRUE, since -90 degrees specifies the same angle as 270 degrees. This can be done as follows:

```
generic
  type LP is limited private;
  with function EQUALS (L, R : LP) return BOOLEAN;
package EQUALITY_OPERATOR is
  function "=" (L, R : LP) return BOOLEAN;
end EQUALITY_OPERATOR;

package body EQUALITY_OPERATOR is
  function "=" (L, R : LP) return BOOLEAN is
  begin
    return EQUALS (L, R);
  end "=";
end EQUALITY_OPERATOR;
```

This generic package can be used to provide a user-defined equality operator for any type:

```
with EQUALITY_OPERATOR;
package POLAR_COORDINATES is
  type POLAR_COORD is
    record
      THETA : INTEGER;
      R      : INTEGER;
    end record;
  function EQUALS (L, R : POLAR_COORD) return BOOLEAN;
package POLAR_EQUALS is
  new EQUALITY_OPERATOR (POLAR_COORD, EQUALS);
  function "=" (L, R : POLAR_COORD) return BOOLEAN
    renames POLAR_EQUALS."=";
end POLAR_COORDINATES;

package body POLAR_COORDINATES is
  function EQUALS (L, R : POLAR_COORD) return BOOLEAN is
  begin
    return (L.THETA mod 360) = (R.THETA mod 360) and
           L.R = R.R;
```

```

        end EQUALS;
    end POLAR_COORDINATES;

    with POLAR_COORDINATES; use POLAR_COORDINATES;
    package USER is
        X : POLAR_COORD := (THETA => -90, R => 5);
        Y : POLAR_COORD := (THETA => 270, R => 5);
        B : BOOLEAN := X = Y;      -- will be TRUE
    end USER;

```

Note that the user-defined equality operator for polar coordinates is even derivable when POLAR_COORD is used as a parent type.

S2. The case statement chooses alternatives based on which alternative contains a choice equal in value to the case expression. Even though the equality operator can be redefined for the scalar type used in a case statement, predefined equality is used to select the case alternative. The redefined equality operation can only be invoked when the equality operator is used explicitly.

S3. The rule for renaming declarations prohibits:

```

type LP is limited private;
function EQUAL (L, R : LP) return BOOLEAN;
function "=" (L, R : LP) return BOOLEAN renames EQUAL; -- illegal

```

EQUAL is not an operator and so the renaming declaration is illegal even though a direct declaration of "=" would be legal.

S4. Although the rule restricting the types for a declaration of "=" allows a renaming declaration to specify an "=" operator with different parameter types, there will be no "=" operation that has such a parameter profile. Hence, the rule, in effect, says that if a renaming declaration declares "=", the parameter types must be the same, but need not be limited.

S5. The overloading resolution rules given in RM 6.6/3 apply to operators as well as to subprograms whose designator is a simple name.

Changes from July 1982

S6. A renaming declaration is allowed for an equality operator even if the parameter types are not limited.

Changes from July 1980

S7. An explicit declaration of the equality operator is now allowed for all limited types, not just limited private types.

S8. A renaming declaration that declares "=" is only allowed to rename another equality operator.

Legality Rules

L1. An operator_symbol used as a designator in a function specification must only be one of the following strings: "and", "or", "xor", "=", "<=", "<", ">=", ">", "+", "-", "&", "not", "**", "/", "mod", "rem", "abs", and "***"; however, any letter in these strings can also be given in upper case (see IG 6.1/S).

L2. The operator_symbols "and", "or", "xor", "=", "<=", "<", ">=", ">", "&", "**", "/", "mod", "rem", and "***" (and their variants using upper case letters) must only be used in function specifications having exactly two parameters.

- L3. The operator_symbols "+" and "-" must only be used in function specifications having one or two parameters.
- L4. The operator_symbols "not" and "abs" (including variants using upper case letters) must only be used in a function specification having a single parameter.
- L5. Operator declarations must not have default values specified for their parameters.
- L6. When "=" is declared by a function declaration, a generic instantiation, or a generic formal subprogram declaration, the formal parameter types must be the same and must be limited.
- L7. When "=" is declared by a renaming declaration, the formal parameter types must be the same, and the designator of the renamed function must be "=".

Test Objectives and Design Guidelines

- T1. Check that "in", "not in", "and then", "or else", "/=", and "!=" are not permitted as operator_symbols. In particular, check that "in" cannot be defined as a function with three parameters of the same type and that "!=" cannot be defined as a procedure with one out and one in parameter.

Check that operator symbols cannot have leading or trailing spaces.

Check that functions for "and", "or", "xor", "=", "<=", "<", ">=", ">", "&", "and", "mod", "rem", and "*" cannot be declared with one or three parameters.

Check that functions for "not" and "abs" cannot be declared with two parameters.

Check that functions for "+" and "-" cannot be declared with zero or three parameters.

Check that default expressions cannot be specified for operator symbol functions.

Check that the parameter types for "=" cannot be different for subprogram declarations, generic instantiations, and generic formal subprogram declarations.

Check that except in renaming declarations (see T5), both parameters in a declaration of "=" cannot have the same scalar, access, or (nonlimited) private type.

Check that except in renaming declarations (see T5), the type of the parameters for "=" cannot be an array type whose components have a scalar, access, (nonlimited) private, or nonlimited record type.

Check that except in renaming declarations (see T5), the type of the parameters for "=" cannot be a record type none of whose subcomponents have a limited type.

Implementation Guideline: For all the above cases, include operators declared by generic instantiation.

- T2. Check that all the permitted operator_symbols can be used in function specifications with the required number of parameters:

- with two parameters -- "and", "or", "xor", "=", "<=", "<", ">=", ">", "&", "and", "mod", "rem", "*", "+", "-.

- with one parameter -- "not", "abs", and "in".

Implementation Guideline: Vary the case of letters in the operator symbols.

Implementation Guideline: Include function declarations, generic instantiations, and generic formal subprogram specifications. (Renaming declarations are tested in T5.)

Check that these functions are invoked when the appropriate operator_symbol is used in an expression.

Check that except for "=" (see T1), operator specifications can have parameters with different types.

- T3. Check that operators for the predefined types can be redefined, e.g., try redefining "+" with

INTEGER arguments and returning an INTEGER result. Check that the redefined operator is invoked when infix notation is used.

- T4. Check that if a renaming declaration declares "=", the renamed function cannot be "/=", nor can it be a function denoted by a simple name or an operator symbol other than "=".

Implementation Guideline: In particular, check that even if the renamed function is a simple name that renames an equality operator, the simple name cannot be used when a renaming declaration declares "=".

Check that when a generic formal subprogram parameter is "=", the default name need not be or denote an equality operator.

- T5. Check that a declaration of "=" need not have parameters of a limited type in a renaming declaration.

Check that when "=" is redefined for a scalar type, a case statement using the scalar type still chooses alternatives based on the predefined equality operation.

Chapter 7

Packages

7.1 Package Structure

Semantic Ramifications

S1. It is a consequence of the syntax (RM 3.9/2 and RM 3.1/2) that a package body, task body, subprogram body, or body stub cannot appear in a package specification. Any such constructs must appear, if at all, in the corresponding package body.

S2. A package specification requires a package body if the specification declares a subprogram, generic subprogram, or task, or if it declares a package or generic package that requires a body. In addition, a package body is required if an incomplete type declaration occurs immediately within the private part of a package and no full declaration is given later in the private part (RM 3.8.1/3).

S3. Visibility rules for packages are discussed in IG 8.3.f/S.

Changes from July 1982

S4. There are no significant changes.

Changes from July 1980

S5. An exceptions part in the body (starting with the reserved word `exception`) is permitted only if it contains at least one exception handler.

S6. Representation specifications are permitted in both the visible and private parts of the package. They are no longer required to follow all other declarations in the private part or visible part.

S7. The remaining textual changes are not substantive. In particular, although generic package declarations and instantiations are no longer `package_declarations`, wherever `package_declaration` appeared in the 1980 syntax, the appropriate terms for a generic package declaration and instantiation were added in 1983. Finally, the July 1982 syntax explicitly shows that the declarative part of a package body may be empty; this was also the case in 1980, since `declarative_part` could itself be empty.

Legality Rules

L1. If an identifier is present at the end of a package specification or package body, it must be the same as the package identifier (RM 7.1/3).

L2. A package body must be provided if any of the following are given as a declarative item of a package specification (RM 7.1/4 and RM 3.8.1/3):

- a (nongeneric) subprogram declaration or a generic subprogram declaration, unless the subprogram is named in an `INTERFACE` pragma accepted by the implementation (see RM 13.9/3),
- a task declaration,
- a (nested) `package_declaration` or a generic package specification that requires a body,
- an incomplete type declaration in the private part of a package without a corresponding full type declaration in the same private part (RM 3.8.1/3).

Exception Conditions

The exceptions that can be raised when elaborating a package specification or a package body are those raised when elaborating any declarative part or those that are propagated (see RM 11.4.1(c)) from the package body.

Test Objectives and Design Guidelines

- T1. Check that if an identifier is present at the end of a package specification or body, it must be the same as the package identifier.

Implementation Guideline: Check for generic and nongeneric packages.

Check that package bodies, subprogram bodies, task bodies, generic unit bodies, and body stubs cannot appear in the visible or private part of package specifications.

Implementation Guideline: The package bodies should all appear at the end of the set of declarative items of either the visible or private part, and in the case of package and task bodies, a package and task specification should appear prior to the body declarations but not interleaved with them.

- T2. Check that all degenerate syntactic cases of package specifications and package bodies are treated correctly. In particular, that both may be empty, that all declarative parts may be empty, and that in a package body, an exception reserved word is permitted only if followed by at least one exception handler. (Note: checks using exception handlers are provided elsewhere; see 11.4/T1, 3, 7, and 11.)
- T3. Check that a package body is provided when it is required (see IG 7.3/T1).
- T4. Check that all forms of declaration (see IG 3.9) are permitted in the private part of a package specification except for deferred constant declarations, private type definitions, package bodies, task bodies, and subprogram bodies (including body stubs for subprograms).

7.2 Package Specifications and Declarations

Semantic Ramifications

S1. Subprogram specifications, package specifications, task specifications, object declarations, etc., can all appear in the private part. Of course, such entities are only known within the private part and the corresponding package body (except for the full declarations of deferred constants, which are declared in the visible part of the package and are thus also visible outside the package.)

S2. An incomplete type declaration (see RM 3.8/3) appearing in the *visible* part of a package must have a complete declaration given later and in the same visible part. If it is given in a *private* part, however, its full declaration must occur later and immediately within either the same private part or the declarative part of the corresponding package body (see IG 3.8.1/S).

S3. A deferred constant declaration can appear only in the visible part of a package specification. Its type must be a private type declared previously in the same package specification (see RM 7.4/4).

S4. RM 3.4/15 states that if a derived type is declared immediately within the visible part of a package, then within this visible part, the type cannot be used as the parent type of a derived type definition. For example:

```
package P is
  type T1 is range 1 .. 10;      -- derived from an integer type
  type NT1 is new T1;           -- illegal
```

```

private
    type NNT1 is new T1;           -- legal in private part
end P;

```

Moreover, RM 7.4.1/4 states that the name of a private type cannot be used in a derived type definition until after the end of the corresponding full type declaration of the private type.

```

package P is
    type T is private;
    type NT1 is new T;           -- illegal
private
    type NT2 is new T;           -- illegal
    type T is range 1 .. 10;
    type NT3 is new T;           -- ok
end P;

```

S5. RM 3.4/11 also states that a derivable subprogram becomes derivable at the end of the visible part in which it is declared. As a consequence, derivable subprograms may be derived both in the private part of the same package and in the package body, as well as outside the package.

```

package P is
    type T1 is private;
    type T2 is private;
    procedure Q (X : T1);
private
    type T1 is ...;
    type T2 is new T1;           -- Derives Q(X : T2); the derived Q
                                -- is not visible outside P.
end P;

package body P is
    procedure Q (X : T1) is ... end Q;
    type T3 is new T1;           -- Derives Q(X : T3) from T1 since Q(X : T1)
                                -- is declared in P's visible part.
    type T4 is new T2;           -- Does not derive Q since Q(X : T2) is
                                -- declared in private part.
end P;

```

Changes from July 1982

S6. There are no substantive changes to this section of the RM. However, changes in RM 3.8.1 affect the treatment of incomplete types in packages (see IG 3.8.1/S).

Changes from July 1980

S7. There are no substantive changes to this section of the RM. However, changes in RM 3.4 affect the treatment of derived types in packages.

Legality Rules

L1. The package identifier must not be a homograph of another declaration occurring immediately within the same declarative region (RM 8.3/17).

Test Objectives and Design Guidelines

T1. Check that a package specification can be declared in a package specification, within either the visible or the private part.

Implementation Guideline: Don't use subprogram or task declarations within either package specification (this situation is tested below in 7.3/T1).

Check that a package body can be provided for a package specification that does not contain any subprogram or task declarations and that statements within the package bodies can be used to initialize variables visible within the package body.

- T2. Check that declarative items in a package specification are elaborated in order.

Implementation Guideline: Use object declarations initialized with functions, or objects having default values provided by functions.

- T3. Check that if an incomplete type declaration is given in the visible part of a package, the full declaration cannot be given within the package's private part or package body, nor can it be given in the visible part of a nested package specification (see IG 3.8.1/T1).

Check that if an incomplete type declaration is given in the private part of a package, the full declaration must be given later, either in the same private part or in the package body (see IG 3.8.1/T8).

In a package which declares a private type T, check that an incomplete type declaration of T may not appear immediately within the visible part, private part, or body of the package.

- T4. Check that only basic declarative items can appear in a package specification, i.e., check that subprogram bodies, task bodies, package bodies, and stubs may not appear either in the visible part or in the private part (see IG 7.1/T1).

- T5. Check that if a derived type is declared in the visible part of a package, it may not be used as a parent type in a derived type definition in the same visible part (see IG 3.4/T17).

Implementation Guideline: Include a numeric type definition as one case of derived type.

Check that other types (i.e., neither private nor derived) may be declared and used as parent types in the same visible part (see IG 3.4/T17).

7.3 Package Bodies

Semantic Ramifications

S1. Since according to RM 7.1/4, the body of any subprogram, package, generic unit, or task declared in a package specification must appear in the corresponding package body, the following kinds of nestings can arise:

```
package A is
  C : constant INTEGER := 6;
  E : constant INTEGER := 4;
  procedure AP (X : INTEGER := C);

  package B is
    D : constant INTEGER := 5;
    procedure BP (X : INTEGER := C + D + E);
  end B;
end A;

package body A is
  D : INTEGER;

  package body B is
    E : INTEGER;
```

```

        procedure BP (X : INTEGER := C + D + A.E) is
            ...
        end BP;
    end B;

```

```

        procedure AP (X : INTEGER := C) is ... end AP;
    end A;

```

This example illustrates that:

- the ordering of declarations in the body need not duplicate the ordering of the declarations in the specification;
- the nesting of package specifications must be duplicated by the nesting of the corresponding package bodies (see RM 7.1/4);
- the visibility rules for identifiers ensure that the default parameter initialization in BP's body declaration refers to the same entities that BP's specification referred to, since in attempting to determine what D denotes, one first looks in B's package body and B's package specification, then in A's package body, and then in A's specification (see RM 8.3). This sequence means that if A.E had been written as E, it would have been associated with a different entity than its association in the initial declaration of BP. Hence, a selected component form of name is needed to ensure that the later specification of BP conforms to the earlier specification (see RM 6.3.1/5).

S2. Note that a null package body can have a semantic effect. If a task object is declared in a package specification, the task is activated after elaborating the declarative part of the corresponding package body (RM 9.3/2):

```

task type T;

X : INTEGER := 1;

function F return INTEGER is
begin
    X := X + 1;
    return X;
end F;

task body T is
    X : INTEGER := F;
begin
    null;
end T;

package P is
    TSK : T;
end P;

package body P is
end P;
-- P.TSK activated

```

```

package CHK is
  B : BOOLEAN := X = 2;    -- must be TRUE
end CHK;

```

Note that package body P is optional. If it is not provided, a virtual body will be assumed to occur at the end of the declarative part.

Changes from July 1982

S3. There are no significant changes.

Changes from July 1980

S4. There are no significant changes.

Legality Rules

- L1. A package body must be provided if any of the following are given as a declarative item of a package specification (RM 7.1/4):
- a (nongeneric) subprogram declaration or a generic subprogram declaration, unless the subprogram is named in an INTERFACE pragma which is accepted by the implementation (see RM 13.9/3),
 - a task declaration (RM 9.1/1),
 - a (nested) package_declaration or generic package specification that requires a body,
 - an incomplete type declaration in the private part of a package without a corresponding full type declaration in the same private part (RM 3.8.1/3).
- L2. The declaration of a package specification in a declarative part must precede the declaration of the corresponding body (see RM 3.9/9).

Test Objectives and Design Guidelines

- T1. Check that if a subprogram or task specification is provided in a package specification, a package body must not be omitted (see also IG 9.1/T3) (if no pragma INTERFACE has been specified for the subprogram).

Check that if a generic subprogram specification is given in a package specification, a package body must not be omitted.

Check that a package body must be provided if an incomplete type is declared in the private part of a package, but no full declaration for the type is given later in the same private part.

Implementation Guideline: Include a check for separately compiled bodies and bodies given as subunits.

Given a package specification Q declared within a package specification P, check that P must have a body containing a body for Q if Q requires a body, i.e., if Q declares:

- a subprogram, task, or generic subprogram;
- a package specification or generic package specification containing either a subprogram, task, or generic subprogram declaration;
- an incomplete type whose full declaration is not given in Q's specification.

Implementation Guideline: Some of these declarations should be given in the private part.

Implementation Guideline: Check for generic and nongeneric packages that require bodies.

- Check that if a subprogram is declared in a package specification by a renaming declaration or by a generic instantiation, no package body is required.
- T2. Check that the statements in a package body are executed after the declarations in the body have been elaborated.
 - T3. Check that exceptions raised in the declarative or statement part of a package body are propagated properly (see IG 11.4/T3, /T7, and /T11).
 - T4. Check that entities declared in a package body cannot be accessed from outside the package body.
 - T5. Check that if a null package body is provided, any tasks declared in the package specification are activated (see IG 9.3/T1).
 - T6. Check that a package body cannot be provided in a declarative part unless its specification has already been declared.
 - T7. Check that the order of declarations (for subprograms, tasks, packages, etc.) which are required in the package body does not need to be the same as the order of their declaration in the specification.

7.4 Private Type and Deferred Constant Declarations

Semantic Ramifications

S1. RM 7.4/4 states that deferred constants are permitted only for private types declared in the same private part as the deferred constant. Hence,

C : constant T;

is illegal unless a declaration for T appears as a declarative item in the same visible part containing the declaration of C. In particular, the following would be illegal:

```

package P is
  type T is private;
  package Q is
    type T is private;
    C : constant P.T;    -- illegal; not in same visible part
  private
    type T is ...;      -- legal, but Q.T is different from P.T
    C : constant T;      -- illegal; must not be in private part
  end Q;
  C : constant Q.T;      -- illegal; Q.T is not a declarative
                        -- item of package P
private
  type T is ...;
end P;
```

S2. When determining the type of an object, it is always important to know whether the object is being referred to inside the package body that knows the definition of the private type. For example,

AD-A189 647

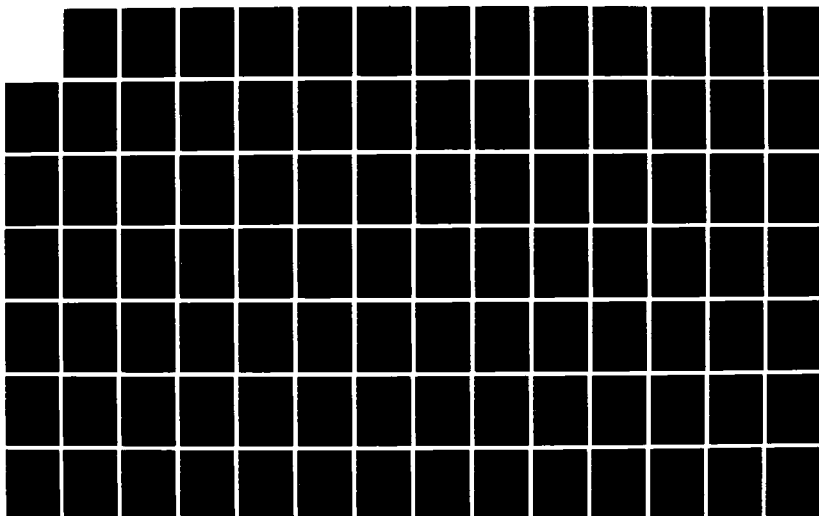
THE ADA (TRADE NAME) COMPILER VALIDATION CAPABILITY
IMPLEMENTERS' GUIDE VERSION 1(U) SOFTECH INC WALTHAM MA
J B GOODENOUGH DEC 86

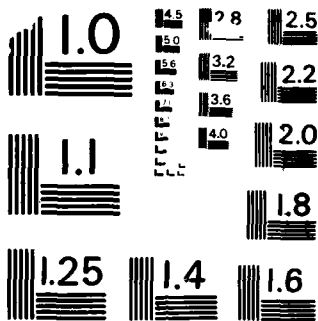
5/9

UNCLASSIFIED

F/G 12/5

NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A


```

package P is
    type T is private;
private
    type T is new INTEGER; -- (1)
end P;

Y : P.T;

package Q is
    type T_NAME is access P.T;
    X : T_NAME;
end Q;

package body P is
begin
    Q.X := new P.T' (3); -- legal (2a)
    Y := 1; -- legal (2b)
end P;

package body Q is
begin
    Q.X := new P.T' (3); -- illegal (3)
end Q;

```

Conversion from *universal_integer* to type T is implicitly declared after the declaration at (1). This implicit conversion operation allows integer literals to be used in contexts requiring a value of type T. The scope of this operation includes P's package body; hence, the allocator and literal written at (2a) and (2b) are legal. However, inside Q at (3), T is private, and so, the scope of implicit conversion from *universal_integer* to type T does not include Q's package body. Hence, no integer literals can be written as though they were values of type T.

S3. All operators (except inequality; see RM 6.7/4) can be overloaded or redefined to accept operands of a limited or nonlimited private type either within the package defining the private type or outside it (see IG 6.7/S). If the redefinition occurs within the package defining the private type, then the subprogram implementing the operation has access to the representation of the private type. If outside this package, however, the subprogram has the usual knowledge of a private type's representation, namely, no knowledge at all (except of discriminants, if any).

Changes from July 1982

S4. There are no significant changes.

Changes from July 1980

S5. A deferred constant declaration must be declared with a type mark; no explicit constraint is allowed in the deferred constant declaration.

Legality Rules

- L1. A private type_declaration used as a declarative_item must appear in the visible part of a package specification; it is not allowed in a subprogram body, task body, block, package body, or the private part of a package specification. (Note: a private type_declaration is also allowed as a generic parameter; see RM 12.1/2.)
- L2. If a deferred constant declaration is given, it must appear in the visible part of a package specification containing a corresponding private type declaration (RM 7.4/4), and the full

declaration of the constant must appear in the private part of this package specification (RM 7.4.3/1).

Test Objectives and Design Guidelines

- T1. Check that a private type declaration (both limited and nonlimited) cannot appear in the private part of a package specification, in a package body, subprogram body, block, or task body.

Check that the full declaration of a private type (limited and nonlimited) cannot appear in the same visible part as does the private type declaration, nor can it be omitted entirely from the package specification (even if it is provided in the corresponding package body).

Check that a full declaration of a private type cannot appear in the private part of a nested package specification.

Implementation Guideline: Check both generic and non-generic packages.

- T2. Check that a deferred constant declaration may only be given in the same visible part as its private type (see IG 7.4.3/T1).

Check that a deferred constant cannot be given for a composite type having a component of a private type (see IG 7.4.3/T1).

Check that a full declaration for a deferred constant must be given in the private part and must have an explicit initialization (see IG 7.4.3/T1), even if the private type has default values specified for all its components.

- T3. Check that no constraint is allowed in the declaration of a deferred constant.

- T4. Check that within the package body, operations depending on the full declaration of a private type are available, but that outside the package body, operations that depend on the characteristics of the full type declaration are not available unless explicit overloads of these operators have been provided by the user.

- T5. Check that all forms of declaration (see IG 3.9) are permitted in the private part of a package specification except for deferred constant declarations (see IG 7.4.3/T1), private type definitions (see T1), package bodies (see IG 7.1/T1), task bodies (see IG 7.1/T1), and subprogram bodies (see IG 7.1/T1) (including body stubs for subprograms) (see IG 7.1/T4).

7.4.1 Private Types

Semantic Ramifications

S1. In this section, we discuss those properties of private types that are common to limited and nonlimited private types. The only difference between the two kinds of private types is that limited private types do not have the operations of assignment and equality implicitly declared; nonlimited private types do.

S2. RM 7.4.1/4 states that a name denoting a private type may not be used in a derived type definition until after the end of the private type's full type declaration. This restriction applies also to a name denoting a subtype of the private type, and to a name denoting a composite type or subtype with a subcomponent of the private type. For example:

```
package P is
  type T1 is private;
  type T2 is array (1 .. 2) of T1;
  type NT2 is new T2;           -- illegal
```

```

        subtype ST2 is T2;
        type NST2 is new ST2;           -- illegal
    private
        type NT3 is new T1;             -- illegal
        type NT4 is new T2;             -- illegal
        type T1 is range 1 .. 2;
        type NT5 is new T1;             -- legal after full declaration
    end P;
```

S3. The identifier used to denote a private type can be used prior to the full declaration of the private type to denote some other entity, e.g.,

```

    type T is private;
    type R (T : INTEGER) is
        record null; end record;
    X : R(T => 3) := (T => 3);          -- legal use of T
```

S4. The full type declaration of a nonlimited private type cannot be a limited type. Thus the full declaration cannot declare a task type, a type derived from a limited type, nor a composite type with a subcomponent of a limited type (see RM 7.4.4/2). For example:

```

package P is
    type T is private;
    task type U is ...;
private
    type T is
        record
            X : U;          -- illegal
        end record;
end P;
```

The full declaration of T is illegal since it contains a component of a limited type (in particular a task type).

S5. If a private type declaration does not include a discriminant part, the type declared by the full type declaration must not be an unconstrained type with discriminants nor an unconstrained array type. The full type declaration can declare a constrained type with discriminants only when the full type is derived from:

- an unconstrained type with discriminants and a discriminant constraint is given in the subtype indication, or
- a previously constrained type with discriminants (in which case no further constraint is permitted).

For example:

```

type BULK_STORAGE is new PERIPHERAL(DROM);    -- see RM 3.7.3/6, /7

package P is
    type T1 is private;
    type T2 is private;
private
    type T1 is new PERIPHERAL(DISK);           -- legal
    type T2 is new BULK_STORAGE;               -- legal
end P;
```

Note that in these instances no component selection operations for the discriminants are implicitly declared in the visible part of the package for the private type (see RM 7.4.2/1), and therefore these operations are not available outside of the package.

s6. For both private and limited private types having discriminants, the only allowed variation in lexical structure between the private type_declaration and the full declaration is that simple names may be replaced by expanded names (and vice versa) if both refer to the same declaration, string literals that denote operators may have upper or lower case characters, and different numeric literals are allowed if they have the same value (see RM 6.3.1). The ability to use expanded names means that selected component notation can be used to resolve naming difficulties, e.g.,

```
X : INTEGER := 5;
package R is
  type T(A : INTEGER := "mod"(X, 3)) is private;
  X : INTEGER := 5;
private
  type T(A : INTEGER := "MOD"(P.X, 03)) is ...;
end R;
```

Note that according to RM 8.3, it is necessary to write P.X in the full declaration of T, since X by itself would refer to R.X. Even though R.X and P.X have the same values at the time both declarations of T are elaborated, the second declaration would be illegal without writing P.X, since it would refer to a different X than the one referred to in the first declaration.

s7. Within the scope of the full type declaration of a private type, the type is not considered "private." Moreover, if the private type is a limited private type and the full type declaration does not declare a limited type, then within the scope of the full declaration the type is not considered "limited." Consider the following example:

```
package A is
  type LP is limited private;
  type P1 is private;
  type P2 is private;
  type P3 is private;
  function F return LP;
  function "+" (L, R : LP) return LP;
private
  type P1 is array (1 .. 2) of LP;           -- illegal
  type LP is new INTEGER;
  type P2 is array (1 .. 2) of LP;           -- legal
  type P3 is new LP;                         -- legal
  function "=" (L, R : LP) return BOOLEAN;  -- illegal
  C1 : BOOLEAN := LP'CONSTRAINED;           -- illegal
  C2 : BOOLEAN := P2'CONSTRAINED;           -- illegal
  procedure G (X : P3 := F + 2);             -- legal
end A;
```

The full declaration of P1 is illegal since it declares a limited type. The full declarations of P2 and P3 are not illegal since LP is not a limited type at the point of the declarations. The redeclaration of equality is illegal since LP is no longer limited. The uses of CONSTRAINED are illegal since LP and P2 are no longer private; moreover, the full declaration does not declare a type with discriminants or a private type. Procedure G is legal since both F and "+" are derived from LP.

s8. A full declaration of a private type must be either a full_type_declaration or a task-

_declaration (for a limited private type). It cannot be an incomplete type declaration nor a subtype declaration (RM 7.4.1/1).

Changes from July 1982

S9. If a private type declaration does not have a discriminant part, the full type declaration may declare a type with discriminants if the type is constrained.

S10. Prior to the end of the full declaration of a private type, the private type — or a name that denotes a type defined in terms of the private type — may not be used in a derived type definition.

Changes from July 1980

S11. If a private type declaration has a discriminant part, the full type declaration must be a record type declaration.

S12. If a name denotes a private type or any type defined in terms of a private type, then before the full declaration of the private type, the only allowed uses of the name are in a deferred constant declaration, a type or subtype declaration, a subprogram specification, or an entry declaration; moreover, the name may not be used within an expression.

S13. The conformance rules in RM 6.3.1 now clarify and change the permissible variations between the discriminant part of a private type declaration and its full declaration.

S14. See also IG 7.4.2/Changes.

Legality Rules

- L1. The full declaration of a private or a limited private type declared in the visible part of a package specification must appear in the private part of the same package specification. It must not be given in a nested package specification.
- L2. If a private type_declaration has a discriminant part, the corresponding full declaration must have a conforming discriminant part (see IG 6.3.1/L) and must be a record type definition.
- L3. If the private type_declaration does not have a discriminant part, the full declaration must not declare an unconstrained type with discriminants.
- L4. A full declaration of a private or a limited private type must not declare an unconstrained array type or an unconstrained type with discriminants.
- L5. A full declaration of a private or limited private type must not be an incomplete type declaration or a subtype declaration.
- L6. The full declaration of a nonlimited private type must not declare a task type nor a type derived from a limited type, nor can it declare an array or record type for which assignment and equality are not defined (see IG 7.4.4/S).
- L7. Prior to the end of the full declaration of a private type, the name that denotes the private type may be used only as the type mark in a deferred constant declaration, an access type definition (but not in a formal generic type definition), a component declaration of an array or record type_declaration, a subtype_declaration, a formal parameter_specification of an entry declaration or a subprogram specification, or a result type in a subprogram specification.

The same restrictions apply to the name of a subtype of a private type and to the names of composite types which have a subcomponent of the private type.

Test Objectives and Design Guidelines

T1. Check that a private type may not be (fully) declared as:

- a type derived from a limited private type, a task type, or a composite type having a subcomponent of a limited type,
Implementation Guideline: Include derivation from a generic formal limited private type.
- a record or array type with a component of a limited private type, a task type, or a composite type having a component of a limited type,
- a record or array type for which assignment and equality are not defined even though no subcomponent of the type is limited (see IG 7.4.4/T9).

Implementation Guideline: Check that the above restrictions are enforced even if all the types are declared in the same package.

T3. Check that within the package that declares a private type, T, and before the end of the corresponding full declaration, neither:

- a name denoting the private type (i.e., T),
- a name that denotes a subtype of T, nor
- a name denoting a composite type with a subcomponent of type T or of a subtype of T

may be used:

- in a derived type definition,
- in a simple expression (as the type mark in a conversion, qualification, membership test (AI-00153), or attribute ('SIZE, 'CONSTRAINED, or 'BASE)),
- in a renaming declaration for an object,
- in the subtype indication of an object declaration or constant declaration,
- as a generic actual parameter,
- in a generic formal parameter (object (having mode in or in out), array type, or access type),
- in an allocator,
- in a representation clause.

Check that these uses of names are permitted in the private part and body after the end of the full type declaration.

Check that the identifier T may be used if it does not denote the private type.

Check that all of the above names may be used in a subtype declaration, a type declaration (for a subcomponent of a composite type or an access type), a subprogram specification (as the formal parameter type or result type in a subprogram declaration, generic formal subprogram declaration, or renaming declaration), or an entry declaration (as a formal parameter type).

Implementation Guideline: Check that these requirements are satisfied both by declarations given immediately within the same declarative region and within nested package specifications.

T4. If a private type declaration contains a discriminant part, check that the corresponding full declaration contains a conforming discriminant part and cannot be a derived type with discriminants.

Implementation Guideline: Include derivation from a formal generic type.

- T5. Check that the full type declaration of a private type without a discriminant part may not declare either a record type with unconstrained discriminants or a type derived from such an unconstrained type.

Implementation Guideline: Include derivation from a formal generic type.

Check that the full type declaration may not declare an unconstrained array type or a derived type that is an unconstrained array type.

Check that the full declaration cannot be an incomplete type declaration or a subtype declaration.

Check that the full declaration may be either a derived type declaration where the parent type is a constrained type with discriminants, or a derived type declaration where the parent type is an unconstrained type with discriminants and an explicit discriminant constraint appears after the type mark.

Implementation Guideline: Include a constrained formal generic type.

Check that the full type declaration may be a derived type declaration where the type mark denotes an unconstrained array type if an explicit index constraint appears as part of the subtype indication.

For both cases of a constrained type with discriminants, check that the discriminants are not visible outside the package.

- T6. Check that a full declaration of a private type without discriminants (limited or nonlimited) can be given in terms of any scalar, array, record (including derived record types with discriminants), access type (including an access type with discriminants), or private type.

Check that the full declaration of a limited private type can be a task type (see IG 7.4.4/T6).

- T7. Check that discriminants may not have the same identifier (see IG 8.3.c/T1).

7.4.2 Operations on Private Types

Semantic Ramifications

S1. If a composite type is declared within the same package as a private type and contains components having that private type, then any additional operations for the composite type (i.e., operations that depend on the full declaration of the component type) are declared at the earliest place within the *immediate scope* (RM 8.2/2) of the composite type and after the full declaration of the private type. Such operations may include relational operators for a one-dimensional array, and equality operators for a composite limited type. One of the consequences of the rule concerning the declaration of additional operations is that, if the composite type is declared in an inner package preceding the full declaration of the private type, then the additional operations are not declared until the package body of the inner package, if at all. (If there is no package body, there are no implicit declarations of these operations.) This is the case because the package body of the inner package is the first place after the full declaration in the immediate scope of the declaration of the composite type. The following example illustrates some ramifications of this rule:

```
package P is
  type T is private;
  type U is array (1 .. 2) of T;           -- (1)
```

```

package Q is
  type V is array (1 .. 2) of T;      -- (2)
private
  type W is array (1 .. 2) of T;      -- (3)
end Q;

private
  type T is range 1 .. 10;            -- (4)
end P;

package body P is                      -- (5)
  package body Q is
    ...                               -- (6)
  end Q;
  ...                                 -- (7)
end P;

```

The following comments pertain to points in the example marked by numbered comments above:

1. The only basic operations declared for U are indexing, slicing (since U is a one-dimensional array), aggregates, assignment, conversion, qualification, membership, and array attributes. The only operators declared for U are equality and inequality (since the component type is not limited) and catenation (since the component type is not limited and U is a one-dimensional array).
2. The operations declared for V are the same as those declared for U.
3. The operations declared for W are the same as those declared for U.
4. All integer operations are declared for T. In addition, since U is now known to have a discrete component type and is a one-dimensional array, relational operators are now declared for U. Note that although V is in scope (RM 8.2/2, /3) and is known to be a discrete array, no relational operations are declared for V; the immediate scope of V only starts with Q's package body. Note that W's scope does not include this point, since W is declared in Q's private part.
5. All operations for T and U are visible throughout the body of P, and no additional operations are declared here for V.
6. Relational operations are declared for V and W; these operations are visible only within the body of Q. Note that (6) is the earliest point within the immediate scope of V and W where these operations may be declared.
7. Additional operations for V and W declared at (6) are not visible here since these are declared within the package body of Q. Additional operations for U are still visible here but not outside of P.

S2. RM 7.4.2/8 indicates that the rules for the implicit declaration of additional operations apply also to indexing, component selection, and slicing operations declared for an access type whose designated type is a private type or an incomplete type. The following example shows the consequence of not having these basic operations available outside the immediate scope of the access type:

```

package P is
  type T is private;

```



```

package Q is
    type U is access T;           -- (1)
end Q;
private
    type T is
        record
            A, B, C : INTEGER;
        end record;               -- (2)
end P;

package body P is
    X : Q.U := new T' (1, 2, 3);  -- (3) -- legal
    Z : INTEGER := X.A;           -- (4) -- illegal

    package body Q is
        Z : INTEGER := X.A;       -- (5) -- legal
    end Q;
end P;

```

1. U.all and the allocator operation for U are implicitly declared here.
2. Component selection operations for A, B, and C are declared for type T, but not for type U, even though the designated type of U is known to contain those components; U's immediate scope does not include this point.
3. The allocation and initialization are legal because aggregate formation and qualification operations have been declared for T (and are in scope), and the allocator for U is in scope.
4. The component selection is illegal because no component selection operations for U are declared yet.
5. All additional operations for U are declared upon entering Q's package body, so this statement is legal.

The example illustrates the importance of knowing where operations are implicitly declared.

S3. The following example also shows the importance of understanding what operations are declared at a given point:

```

package P is
    type LP is limited private;
private
    type LP is new INTEGER;
end P;

X : P.LP;
type NLP is new P.LP;    -- derived limited type
procedure R (X : NLP) is ... end;

package body P is
    XNLP : NLP;
begin
    X := 1;               -- (1) legal
    XNLP := 1;            -- (2) illegal

```

```

R(1);          -- (3) illegal
R(NLP(LP(1))); -- (4) legal
end P;

```

(1) is legal since assignment is declared for LP when LP is fully declared, and hence is available in P's body. Similarly, conversion from *universal_integer* to LP is also declared when LP is fully declared; this conversion operation allows the use of integer literals in contexts requiring values of type LP (RM 4.6/15).

(2) is illegal because neither assignment nor conversion from *universal_integer* is declared after NLP's declaration, since the parent type is a limited type (RM 3.4/5). However, conversion from NLP to LP, and vice versa, is declared (RM 3.4/5).

(3) is illegal because there is no conversion from *universal_integer* to NLP, but (4) is legal because such a conversion is declared for type LP, and conversion to NLP's parent type is declared after NLP's declaration.

S4. The rules for hiding of implicitly declared operations (see RM 8.3/17) need to be given special attention in packages. The rules state that immediately within the same declarative region, two declarations must not be homographs unless exactly one of them is an implicit declaration (of a predefined operation or a derived subprogram). When these conditions are met, the following relationships hold:

- an explicit declaration always hides an implicit declaration, regardless of which appears first,
- an implicit declaration of a derived subprogram always hides an implicit declaration of a predefined operator, regardless of which appears first.

In the case of inequality (which is never explicitly declared), the rules in RM 8.3/17 specify that hiding of inequality operations follows exactly the same pattern as hiding of the corresponding equality operations.

The following two examples illustrate some of the ramifications of this rule and its interactions with derived types (see RM 3.4/11).

```

package P1 is
  type T1 is range 1 .. 10;
  -- implicit declaration of integer predefined operations,
  -- namely "+", "-", etc., and "<", "=", etc.
  C1: constant T1 := 2;
  D1: constant T1 := C1 + C1;  -- operation is predefined "+"
  function "+" (L, R : T1) return T1;
  -- Redeclaration hides predefined "+". New "+" is visible for
  -- rest of package specification, package body, and outside of
  -- package, and is a derivable subprogram.
private
  function "-" (L, R : T1) return T1;
  -- redefinition hides predefined "-" for rest of package and
  -- body but is not visible outside and not derivable.
end P1;

use P1;

```

```

package P2 is
  type T2 is private;
  function "+" (L, R : T2) return T2;
  function "*" (L, R : T2) return T2;
  -- both "+" and "*" are derivable subprograms
private
  function "/" (L, R : T2) return T2; -- not derivable
  type T2 is new T1;
  -- All arithmetic operators derived from T1 are implicitly
  -- declared here, namely user-defined "+", predefined "*",
  -- "-", "<", etc. Even though these implicit declarations
  -- occur later than the explicit declarations, the implicit
  -- declarations of "+", "*", and "/" are hidden according to the
  -- rules of RM 8.3/17.
end P2;

```

The next example illustrates derivation and hiding of the equality operator for limited private types (see also RM 7.4.4):

```

package LIMITED_TYPES_1 is
  type LT1 is limited private;
  function "=" (L, R : LT1) return BOOLEAN;

  type LT2 is limited private;
  -- no "=" operation declared here for LT2
private
  type LT1 is range 1 .. 10;
  -- predefined "=" is hidden by explicit declaration
  LC1, LC2 : constant LT1 := 2;
  NO_GOOD : constant BOOLEAN := LC1 = LC2;
  -- raises PROGRAM_ERROR since body of "=" not elaborated

  type LT2 is range 1 .. 10;
  -- predefined "=" visible only for rest of package and body,
  -- no "=" for LT2 visible outside of package.
end LIMITED_TYPES_1;

use LIMITED_TYPES_1;

package LIMITED_TYPES_2 is
  type LT3 is limited private;
  type LT4 is limited private;
  type LT5 is new LT1; -- derives "=" from LT1
  type LT6 is new LT2; -- no "=" derived or declared
  type T7 is private; -- note: does not say "limited"
private
  type LT3 is new LT1;
  -- derives "=" from LT1, though only visible inside package
  type LT4 is new LT2; -- no "=" derived or declared
  type T7 is new LT1;
  -- illegal, full type may be a limited type only if private
  -- type declaration says "limited"
end LIMITED_TYPES_2;

```

S5. The attributes T'BASE and T'SIZE are declared for private types, although they may not be used within the package until after the end of the full declaration of the private type (see RM 7.4.1/4). T'CONSTRAINED also cannot be used within the package before the full type declaration, and it cannot be used *after* the full declaration either, unless the full declaration derives from a private type. In addition, if the private type declaration has a discriminant part, 'CONSTRAINED is permitted for objects of the private type both inside and outside the package (RM 3.7.4/2). Note that there is, in effect, a special rule for 'CONSTRAINED: although 'CONSTRAINED is declared after the declaration of a private type, it is not usable after the full type declaration even though visible, unless the full type declaration implicitly declares a new 'CONSTRAINED operation:

```
package P is
  type T1 is private;
  type U (D : INTEGER) is
    record null; end record;
private
  type T1 is new U(1);           -- T1 has discriminants
  A : T1;
  B : BOOLEAN := T1'CONSTRAINED; -- illegal; T1 not private
  C : BOOLEAN := A'CONSTRAINED;  -- legal; A has discriminants
end P;

package Q is
  type T2 is private;
private
  type T2 is new P.T1;
  A : T2;
  B : BOOLEAN := T2'CONSTRAINED; -- legal; T2 is private
  C : BOOLEAN := A'CONSTRAINED;  -- illegal; A has no discriminant
end Q;
```

S6. Note that it is possible to use 'CONSTRAINED prior to the full declaration of the private type. For example:

```
package P is
  type T (D : INTEGER) is private;
  C : constant T;
  procedure G (X : BOOLEAN := C'CONSTRAINED); -- see RM 7.4.3/2
private
  ...
end P;
```

S7. T'CONSTRAINED exists primarily for generic formal private types. It allows the body of a generic unit to provide different handling for objects of a formal type depending on whether the objects all have the same constraints (and thus size) or possibly differing constraints (and thus varying sizes). For example, the generic body for the SEQUENTIAL_IO package (see RM 14.2) might use this attribute to decide whether fixed-size (external) records or varying-size records are appropriate for the external file.

S8. Note that 'CONSTRAINED is only defined for private types and subtypes. In particular, it is not allowed for composite types containing a subcomponent of a limited type.

S9. User-defined operations on limited private types and on nonlimited private types may be declared outside the package that declares the type, as well as inside. Any operations declared outside, however, have no special access to the representation of the private types.

S10. Apart from the attributes discussed above ('SIZE, 'BASE, and 'CONSTRAINED), the only operations implicitly declared for private types are the following:

- the operations involved in assignment (except for limited private types),
- predefined equality and inequality (except for limited private types),
- membership tests,
- qualification and conversion, and
- component selection for any discriminants declared in the private type declaration.

Since the name of the type may not be used in a simple expression prior to the end of the full declaration (see RM 7.4.1/4), membership tests, qualification, conversion, and attributes may not be used until after the end of the full declaration.

Changes from July 1982

- S11. The attribute 'SIZE is available for every private type.
- S12. RM 7.4.2/4 now says all operations declared within a visible part are available outside the package (not just operations having a parameter of a private type declared in the visible part).
- S13. The rule (RM 7.4.2/6) for operations implicitly declared for composite types is extended to include any composite type with a *subcomponent* of a private type.
- S14. 'CONSTRAINED is allowed for subtypes of a private type.
- S15. The rules for implicit declarations of additional operations (see S2 and S3 above) apply also to access types whose designated types are private types or incomplete types.

Changes from July 1980

- S16. This section now incorporates information not stated explicitly in the 1980 version. It also incorporates information that existed in other 1980 sections. We describe here (under Changes) only those changes relevant to the contents of the July 1982 section.
- S17. The attribute 'BASE is declared for every private type.
- S18. The attributes 'SIZE and 'ADDRESS are available for objects of every private type.
- S19. 'CONSTRAINED is available for private types without discriminants.
- S20. The operations for private types include membership tests, selected components for discriminants, qualification, and explicit conversion.
- S21. The places where implicit declarations of additional operations occur have been made explicit.

Legality Rules

- L1. If a one-dimensional array type is declared with a component type that is an as-yet incompletely declared private or limited private type, then the following operations are only implicitly declared for the array type (and thus usable) after the full declaration of the component type and within the immediate scope of the array type:
- relational operators, if the component type is a scalar type;
 - logical operators, if the component type is a Boolean type;
 - string literals, if the component type is a character type;

- equality, inequality, catenation, assignment, and aggregates, if the component type was a limited private type and the full declaration declares a nonlimited type.
- L2. If a multi-dimensional array type is declared with a component type that is an as-yet incompletely declared limited private type and the full declaration declares a nonlimited type, then equality and inequality are only implicitly declared for the array type (and thus usable) after the full declaration of the component type and within the immediate scope of the array type.
- L3. If the designated type of an access type is an as-yet incompletely declared private or limited private type, then the following operations are only implicitly declared (and thus usable) after the full declaration of the designated type and within the immediate scope of the access type:
- indexing, 'FIRST, 'LAST, 'LENGTH, and 'RANGE, if the designated type is an array type;
 - slicing, if the designated type is a one-dimensional array type;
 - component selection, if the designated type is a record type with components.
- L4. When T is a type, the attribute $T.CONSTRAINED$ is allowed only for a private type or subtype T , including generic formal private types.

Test Objectives and Design Guidelines

- T1. Check that predefined equality and inequality are implicitly declared for nonlimited private types at the point of the private type declaration, and are not declared for limited private types or for composite types which have a subcomponent of a limited private type.
- T2. Check that operations for arrays of private types which depend on characteristics of the full declaration of the private type are not visible or usable outside the package. (Such operations are relational operations, logical operations, string literals, and, for limited private component types, equality, inequality, catenation, assignment, and aggregates.)
Implementation Guideline: In the limited private case, use an array with a subcomponent of a limited type.
- Check that when the designated type of an access type is a private type, operations for the access type that depend on characteristics of the full declaration of the private type are not visible or usable outside the package. Such operations are indexing, slicing, 'FIRST, 'LAST, 'LENGTH, and 'RANGE when the type is an array type, and component selection when the type is a record type.
- T3. Check that membership tests, qualification, and explicit conversion are available for private types (limited and nonlimited).
- Check that other basic operations are not available, especially those that are available for the full type.
Implementation Guideline: Include a case where a derived private type is visible inside the parent type's package body.
- T4. Check that redeclaration of equality is illegal after the full declaration of a limited private type, if the full declaration does not itself declare a limited type.
- T5. Check that the additional operations for a composite type with a component or subcomponent of a private or limited private type are not declared before the earliest place within the immediate scope of the composite type's declaration and after the full declaration of the private type.

If the designated type of an access type is an incomplete, private, or limited private type whose full declaration has not yet been given, check that the additional operations for the access type are not declared before the earliest place within the immediate scope of the access type's declaration and after the full declaration of the private type.

Implementation Guideline: For incomplete types, include a case where the full declaration is given in the package body.

Implementation Guideline: Include full declarations that derive array types, types with discriminants, and task types.

- T6. Check that component selection is available for any discriminant declared in the private type declaration.

Check that no selection for discriminants is available prior to the end of the full declaration if the full type is a constrained type with discriminants (see IG 7.4.1/T5).

- T7. Check that T'BASE and T'SIZE are available for private type T outside the package declaring T, and for the full type after the end of the full type declaration.

Check that T'CONSTRAINED is available for the private type (limited and nonlimited) outside the package, whether or not the private type has discriminants; check that 'CONSTRAINED is not available prior to the end of the full declaration, and is only available after the full declaration if the full declaration derives from a private type.

Implementation Guideline: Check that the presence of discriminants does not affect the legality of T'CONSTRAINED.

Check that 'CONSTRAINED is not allowed for a composite type containing a subcomponent of a private type.

Check that 'CONSTRAINED can be used for generic formal private types, and that it returns the correct value.

Check that no other attributes are available for private types. In particular, check that attributes that are legal for the full type are not available outside the package.

- T8. Check that A'SIZE and A'ADDRESS are available for objects of private types (limited and nonlimited) both inside and outside the package.

Check that A'CONSTRAINED is available outside the package that declares the private type for objects of a private type with visible discriminants, and is available before and after the full declaration of a private type.

Check that A'CONSTRAINED is only available after the full declaration of a private type that has no discriminants if the full declaration of the type declares a type with discriminants.

Implementation Guideline: Include deriving from a private type with discriminants.

- T9. Check that subprograms that use a private type as a parameter type or as a result type can be declared both inside and outside the package that declares the private type (limited or nonlimited).

Implementation Guideline: Include formal generic subprograms and renaming declarations.

Check that out parameters having a limited private type can be declared in the package declaring the type.

- T10. Check that operator symbols using a private type as a parameter or result type may be declared both inside and outside the package that defines a private type (limited or nonlimited).

Check that an equality operation may be declared for a limited private type (see RM 6.7/4) inside the package that declares it, and that an equality operation may be declared for a limited private type outside the package as well.

- T11. Check that within a package specification and body, any explicit declarations of operators and subprograms hide any operations implicitly declared at the point of the full declaration, whether the explicit declarations precede or follow the implicit declarations.

Check that the hiding rules apply also to redefined equality and inequality for limited private types.

7.4.3 Deferred Constants

Semantic Ramifications

S1. A deferred constant may be given only in the visible part of a package. Its type must be a private type declared in the same visible part (see RM 7.4/4; see also IG 7.4/S). In particular, note that deferred constants cannot be given for composite types having a subcomponent of a private type.

S2. If a deferred constant declaration appears in the visible part of a package, a full declaration of the constant with an explicit initialization must be given immediately within the private part. The full declaration may not be given in the package body or within a nested package (see IG 7.4/S and RM 7.4/4). Note that a full declaration with an explicit initialization is required even if the private type has default values for each component.

S3. The type mark of a deferred constant declaration may differ from that of the full declaration only according to the conformance rules (see IG 6.3.1), i.e., an expanded name may be used for the type mark instead of a simple name.

Since the conformance rules require that the type marks of the deferred and full declaration have "the same meaning" as given by the visibility rules (see RM 6.3.1/5), the names must be associated with the same declaration (see RM 8.3/2). Hence, a name declared with a subtype declaration does not conform to a name declared with a private type declaration:

```
package A is
  type T (D : INTEGER) is private;
  subtype ST is T;
  subtype T3 is T(3);
  CT3 : constant T3;
  CT  : constant T;
private
  type T (D : INTEGER) is
    record null; end record;
  CT3 : constant T(3) := (D => 3);  -- illegal: T and T3 are not
                                   -- declared by same decl
  CT  : constant ST := (D => 3);    -- illegal: T and ST are not
                                   -- declared by same decl
end A;
```

S4. The conformance rules for deferred constants refer only to the type mark, and do not prohibit the full declaration from having a constraint in the subtype indication. If a constraint is given, it must be appropriate for the type mark. If the initial value (which must be present in any case) is not compatible with the constraint, CONSTRAINT_ERROR must be raised. For example:


```

package B is
  type T (D : INTEGER) is private;
  C4, C2, C3, C1 : constant T;
  C5 : constant T(D => 3);      -- illegal; no constraint allowed
private
  type T (D : INTEGER) is
    record null; end record;
  C1 : constant B.T := (D => 1);    -- legal
  C2 : constant T(2) := (D => 2);   -- legal; no CONSTRAINT_ERROR
  C3 : constant T (D => 3);         -- illegal; no initialization
  C4 : constant T(4) := (D => 5);   -- legal; CONSTRAINT_ERROR
end B;

```

s5. Multiple declarations of deferred constants are permitted both for the deferred constant declarations and for the corresponding full declarations. There is no requirement that both declarations be multiple declarations if one is so. There is no requirement, either, that the order of full declarations duplicate the order of the deferred constant declarations (see example above).

s6. RM 7.4.3/2 allows names of deferred constants to appear prior to their full declaration, but only in the default expressions for record components and for formal parameters of subprograms. An attempt to access the value of such deferred constants is erroneous, not illegal; such an attempt does not necessarily raise PROGRAM_ERROR. For example:

```

package P is
  type T is private;          -- 1
  C : constant T;             -- 2
private
  type T is range 1 .. 10;
  function F(X : T := C) renames "+"; -- 3 -- legal use of C
  V : T := C;                 -- 4 -- illegal use of C
  type U (D : T := C) is      -- 5 -- legal, D is a component
    record
      A : T := C-2;          -- 6 -- legal use of C and "-"
    end record;
  R : U;                      -- 7 -- erroneous use of C
  S : INTEGER := F;           -- 8 -- erroneous use of C

  generic
  package GP is
    R : U;                    -- 9 -- nonerroneous use of C
    S : INTEGER := F;         -- 10 -- nonerroneous use of C
  end GP;

  package NGP is new GP;      -- 11 -- erroneous use of C
  type V is range 1 .. C;     -- 12 -- illegal use of C
  C : constant T := 8;
  W : T := C;                 -- 13 -- legal use of C
  package NGP2 is new GP;     -- 14 -- nonerroneous use of C
end P;

```

Point (4) is illegal since the constant C has not yet been fully declared. Note that (5) is legal and (7) is erroneous because only in (7) is the (nonexistent) value of C actually used (in the default initial value). Similarly, (9) and (10) are legal and nonerroneous because these declarations are

not elaborated until the generic unit is instantiated. Since the instantiation at (11) occurs before the full declaration of C, the instantiation is erroneous. For the same reason, the instantiation at (14) is nonerroneous. Finally, note that

```
C : constant T := F;
```

would be erroneous, since "before the elaboration of the corresponding full declaration" means "not after" the elaboration, and since the use of C's value in the invocation of F does not occur after the elaboration of C's full declaration.

Note that because (7) and (8) are erroneous, not illegal, the program cannot be rejected at compile time (RM 1.6/10), but PROGRAM_ERROR may be raised at run-time.

Changes from July 1982

S7. There are no significant changes.

Changes from July 1980

S8. Prior to the end of the full declaration of a deferred constant, a name that denotes the constant may be used only in the default expression of a record component or subprogram formal parameter.

S9. Permissible differences between the declaration of a deferred constant and the constant's full declaration have been clarified in RM 7.4.3 and RM 6.3.1.

Legality Rules

- L1. A full declaration of a deferred constant must appear immediately within the private part of the package declaring the deferred constant (RM 7.4.3/1); the full declaration must have an explicit initialization (RM 3.2.1/2).
- L2. The type mark in a deferred constant declaration must denote a private type that is declared immediately within the same visible part (RM 7.4/4).
- L3. The names used as type marks in a deferred constant declaration and its full declaration must both denote the same declaration (RM 6.3.1/5, 6 and RM 8.3/2).
- L4. Prior to the end of its full declaration, a name that denotes a deferred constant may be used only in default expressions of record components and formal parameters of subprograms.
- L5. A deferred constant declaration is allowed only in the visible part of a package (RM 7.4/4).

Exception Conditions

- E1. PROGRAM_ERROR may be raised by an attempt to use the value of a deferred constant prior to the elaboration of its full declaration (RM 7.4.3/4 and RM 1.6/10).

Test Objectives and Design Guidelines

- T1. Check that a deferred constant declaration cannot be given:
 - for a private type declared in some other package,
 - for a composite type having a component of a private type,
 - in the private part of a package (even the package containing the private type's declaration, whether or not it appears before the full declaration),
 - in the corresponding package body, or

- a package specification nested within the package specification containing the private type declaration.

Implementation Guideline: Check both generic and nongeneric packages.

Check that even if a private type has default values specified for all its components, a full declaration must be given for a deferred constant, and the full declaration cannot omit an initialization expression (see RM 3.2.1/2).

- T2. Check that multiple declarations may be used for deferred constant declarations, even if the full declarations are given individually.

Check that multiple declarations may be used for the full declarations of deferred constants, even if the deferred constant declarations are given individually.

Check that the order of deferred constant declarations need not be duplicated by the order of full declarations, even if one or both groups are given as a multiple declaration.

Check that when the full declaration of a deferred constant is given as a multiple declaration, the initialization expression is evaluated once for each deferred constant.

- T3. Check that the type marks of the deferred constant declaration and the corresponding full declaration must denote the same type or subtype declaration.

- T4. Check that before the end of the full declaration of a deferred constant, the following uses of a name that denotes the deferred constant are illegal.

- in the initialization expression of an object or a constant declaration.

Implementation Guideline: Include usage within an aggregate and as a function's actual parameter.

- as the default expression of a generic formal parameter,
- as a generic actual parameter (for both in and in out parameters),
- within a range expression of a subtype indication (in a type or object declaration after the full declaration of the private type),
- in an index constraint of a subtype indication (of a type or object declaration after the full declaration of the private type),
- as a discriminant constraint of a subtype indication,
- in a renaming declaration.

Implementation Guideline: Check that all the above uses are illegal even if they occur in the scope of the full declaration of the private type of the constant.

Implementation Guideline: Check uses even where the illegal use is not elaborated.

Implementation Guideline: Include uses within the deferred constant declaration itself.

- T5. Check that use of a deferred constant in the default expression of a record component or subprogram formal parameter is permitted.

Implementation Guideline: Include use in a default expression of a generic subprogram, in a formal generic subprogram parameter, and in a renaming declaration.

Implementation Guideline: Include use in an expression containing an operator (after the full declaration of the type); use with an attribute ('SIZE or 'ADDRESS before the full declaration of the private type, and other attributes, e.g., 'FIRST, after the full declaration); and use as a function actual parameter.

- T6. Check that after the full declaration of a deferred constant, the value of the constant may be used in any expression. In particular, check that the uses listed in T4 are permitted.

- T7. Check that an explicit constraint may be given in the subtype indication of the full

declaration of a deferred constant. Check that if a constraint is present, it must be appropriate for the type mark.

Check that no constraint is allowed in a deferred constant declaration (see IG 7.4/T3).

- T8. Check that an attempt to use the value of a deferred constant prior to the end of its full declaration is not illegal.

Check whether an attempt to use the value of a deferred constant prior to the end of its full declaration raises PROGRAM_ERROR.

7.4.4 Limited Types

Semantic Ramifications

S1. RM 7.4.4/1 defines a *limited type* as a type for which predefined assignment and equality are not implicitly declared. RM 7.4.4/2 enumerates some sufficient conditions for a type to be limited, but this list is not complete. In particular, although a composite type is always limited if the type of any component is limited, it is also possible for a composite type to be limited even if the type of *no* component is limited. Consider the following example:

```
package P is
  type LP is limited private;
  type NP is private;
  package Q is
    type LP_ARRAY is array (1 .. 2) of LP;
  end Q;
private
  type LP is new INTEGER;
  type NP is new Q.LP_ARRAY;      -- illegal; LP_ARRAY is limited.
end P;                             -- "=" and "!=" not yet
                                   -- declared for LP_ARRAY.
```

Composite type LP_ARRAY is a limited type until the body of Q, even though no component of LP_ARRAY is limited after the full declaration of LP, because assignment and equality for LP_ARRAY are not implicitly declared until reentering LP_ARRAY's immediate scope (RM 7.4.2/7). The consequences of this rule are further illustrated by considering P's body.

```
package body P is
  use Q;
  function "=" (L, R : LP_ARRAY) return BOOLEAN is -- legal
  begin ... end;

  generic
    type T is private;          -- note: not limited private
  package A is
    ...
  end A;
  package NEW_A is new A (LP_ARRAY); -- illegal; LP_ARRAY limited

  package body Q is
    function "=" (L, R : LP_ARRAY) -- illegal; LP_ARRAY not
    -- limited
    return BOOLEAN is
  begin ... end;
```

```

        package ANOTHER_NEW_A is new A (LP_ARRAY); -- legal
    end Q;
end P;

```

Although the example is somewhat contrived, it is important to keep in mind that "limited" means having no predefined assignment and equality. The rules of RM 7.4.2/7 state that these operations are not declared until the body of Q, and therefore, LP_ARRAY must be treated as a limited type until that point.

S2. If a record has a limited component in a variant part, the record type is nonetheless limited, so aggregates specifying only nonlimited components cannot be written:

```

type T (D : BOOLEAN) is
  record
    case D is
      when TRUE  => C1 : INTEGER;
      when FALSE => C2 : TASK_TYPE;
    end case;
  end record;
TT : TASK_TYPE;
function R (X : T) return BOOLEAN;
X1 : BOOLEAN := R((TRUE, 3));      -- illegal aggregate
X2 : BOOLEAN := R((FALSE, TT));    -- illegal aggregate

```

S3. Note that if "=" is overloaded for a limited private type, T, a composite type (e.g., an array of T's) still does not have equality defined, since equality for composite types is not defined in terms of equality of the component types (see RM 4.5.2/8). Equality can be defined, however, for any limited type (see RM 6.7/4).

S4. Note that assignment and pre-defined equality operations are permitted on limited private types within a package specification as soon as the full declaration of the private type has been given. For example,

```

package P is
  type T is limited private;
  C : constant T;
  D : constant T;
private
  type T is new INTEGER;
  E : constant T := 5;      -- legal now
  C : constant T := 3;
  D : constant T := 4;
  CHECK : BOOLEAN := C = D; -- legal now
end P;

```

S5. If a deferred constant is declared for a limited private type, the full declaration of the type must declare a nonlimited type, since if it declares a limited type, the deferred constant cannot be given a value:

```

package P is
  type LP is limited private;
  C : constant LP;
private
  type LP is new TASK_TYPE; -- derives a limited type
  C : constant LP := ...;    -- illegal: no assignment for LP
end P;

```

S6. An access type whose designated type is a limited type is not itself a limited type. Thus assignment and equality are available for objects of the access type, though still not available for the objects designated by the access values. For example:

```
package A is
  type LP is limited private;
  type ACC is access LP;
  function F return ACC;
  function G return ACC;
private
  type LP is new INTEGER;
end A;

use A;
P1 : ACC := F;
P2 : ACC := G;
X : BOOLEAN := P1 = P2;           -- legal
Y : BOOLEAN := P1.all = P2.all;  -- illegal
```

S7. In an explicit declaration of a subprogram, entry, or generic subprogram, a formal parameter of mode out may have a limited type only under the following circumstances.

- the type is a limited *private* type (i.e., not a composite limited type or a task type),
- the subprogram, generic subprogram, or entry declaration occurs within the visible part of the package that declares the limited private type,
- the full declaration of the limited private type does not itself declare a limited type.

One of the consequences of this rule is that a task type cannot ever be used as the type of an out parameter, nor can a limited private type whose full declaration declares a task type. Another consequence is that, even though user-defined operations on limited private types may be declared outside the package that defines the type, these operations may not have an out parameter of that type. The above restrictions apply also to composite limited types and types derived from limited types; they may not be used as the type of an out parameter. For example:

```
task type TASK_TYPE is ... end TASK_TYPE;
package A is
  type LP1 is limited private;
  type LP2 is limited private;
  type ARR_LP1 is array (1 .. 2) of LP1;
  type NEW_TASK_TYPE is new TASK_TYPE;
  procedure G1 (X : out LP1);           -- 1 -- legal
  procedure G2 (Y : out LP2);           -- 2 -- illegal
  -- full declaration is limited
  procedure G3 (Z : out ARR_LP1);       -- 3 -- illegal
  -- type is not limited private
  procedure G4 (W : out NEW_TASK_TYPE); -- 4 -- illegal
  -- type is not limited private
private
  procedure G5 (X : out LP1);           -- 5 -- illegal
  -- declaration not in visible part
  type LP1 is new INTEGER;
```

```

type LP2 is new TASK_TYPE;

procedure G6 (Y : out LP1);           -- 6 -- legal
    -- LP1 not considered limited here
procedure G7 (Z : out LP2);           -- 7 -- illegal
    -- declaration not in visible part; LP2 not limited private

end A;

```

S8. The restriction forbidding subprogram out parameters of limited types applies only to "explicit subprogram_declarations", entry declarations, and generic procedure declarations. Since a generic formal subprogram is not declared with a subprogram_declaration, this restriction does not apply to formal subprograms, nor does it apply to renaming declarations:

```

generic
    type LP is limited private;
    with procedure P (X : out LP);      -- legal
package P1 is
    procedure S (X : out LP);           -- illegal
    procedure NP (X : out LP) renames P; -- legal
end P1;

```

Note that declaration of a subprogram with an out parameter of a limited private type is not restricted to *immediately* within the package declaring the type. Hence, such declarations can occur in nested packages, even in the private part of a nested package.

S9. The unavailability of assignment for limited types has certain consequences, some of which are indicated in RM 7.4.4/6-8. Explicit initializations for variables, constants, and record components are not allowed if the type is limited, since initialization requires the assignment operation (RM 3.2.1/8). Similarly, an explicit initial value is not permitted in an allocator if the designated type is limited, since an assignment is needed (RM 4.8/6).

The fact that a generic formal parameter of mode in must not have a limited type is not really a consequence of the unavailability of assignment; although RM 12.1.1/3 requires that such generic actuals be copied, the act of making a copy is not, strictly speaking, assignment. Consequently, the restriction is also explicitly stated in RM 12.1.1/3. Note that limited types *are* allowed as subprogram in parameters even when their values must be copied to the formal parameter (e.g., when the full declaration of a limited type declares a scalar or access type; RM 6.2/8), because copying is not, technically speaking, assignment.

Changes from July 1982

S10. In an explicit declaration of a procedure, entry, or generic procedure, a formal parameter of mode out may have a limited type only if:

- the type is a limited private type,
- the declaration occurs within the visible part of the package that declares the limited private type, and
- the full declaration does not declare a limited type.

Changes from July 1980

S11. The attribute 'BASE is declared for every limited type.

S12. The attributes 'SIZE and 'ADDRESS are declared for objects of every limited type.

S13. 'CONSTRAINED is declared for limited private types without discriminants.

S14. The operations for limited types include membership tests, selected components for discriminants, qualification, and explicit conversion.

S15. Operations for a private type (i.e., subprograms with a parameter of a private type) may be declared outside the package that declares the type.

S16. Formal subprogram parameters of a limited type may have default values.

S17. A generic formal parameter of mode In is not allowed to be a limited type.

Legality Rules

L1. The full declaration of a nonlimited private type must not be a limited type (RM 7.4.1/3).

L2. An out parameter of a subprogram, generic subprogram, or entry declaration must not have a limited type unless:

- the type is a limited private type,
- the declaration of the subprogram, generic subprogram, or entry occurs within the visible part of the package that declares the limited private type (including within any nested packages), and
- the full declaration of the limited private type does not declare a limited type.

Test Objectives and Design Guidelines

T1. Check that in the specification of a subprogram declaration, entry declaration, or generic subprogram declaration, a parameter of mode out may not have a limited type if:

- the limited type is a task type, a composite limited type, or a derived limited private type.

Implementation Guideline: Include derivation from a generic formal type.

- it is a limited private type and the declaration of the subprogram, entry, or generic subprogram appears in the private part or body of the package declaring the limited private type, or outside the package declaring the type.

Implementation Guideline: Check that parameters of mode out are allowed in packages nested in the visible part, including within the private part of such nested packages.

- the full declaration of the limited private type declares a task type, a composite type with a limited component, or a type derived from a limited private type.

Check that an out parameter of a limited type is permitted in the private part after the full declaration of the limited private type if the full declaration does not declare a limited type.

Check that a generic formal subprogram parameter can have an out parameter of a limited type.

Check that a renaming declaration can declare a subprogram with an out parameter of a limited type.

T2. Check that a subprogram parameter of a limited private type may have a default expression, even if the subprogram is declared outside the package that declares the private type.

Check that a subprogram parameter may have a default expression if its type is a composite type with a limited private component or a type derived from a limited private type, even if the subprogram is declared outside the package that declares the limited type.

T3. Check that a variable declared outside the package defining a limited private type cannot have an initialization specified explicitly (see IG 3.2.1/T1).

- T4. Check that variables and constants of a limited private type cannot be compared for equality (or inequality) outside the package defining the type, nor can they be assigned to or given initial values.

Implementation Guideline: Include an access type whose designated type is limited.

Implementation Guideline: Include instantiation of a package that declares a limited type.

Check that the restrictions on use of assignment and equality extend to objects having at least one component of a limited private type, and to types derived from limited private types.

- T5. Check that a generic *In* parameter may not be a limited type (see IG 12.1.1/T4).
- T6. Check that the full declaration of a limited private type can declare a task type, a type derived from a limited private type, and a composite type with a component of a limited type (see also T9).
- T7. Check that constants and variables of a limited private type can be declared and initialized in the private part and body of a package after the full declaration of the private type, if the full declaration is not limited.
- Check that pre-defined equality and assignment is defined and available within the private part and body after the full declaration of the private type if the full declaration is not limited.
- T8. Check that use of user-defined equality for a limited private type within the private part of the package declaring the type raises `PROGRAM_ERROR` (since the body of the equality operator has not yet been elaborated; see IG 3.9/T6).
- T9. Check that if a composite type is declared in the same package as a limited private type and has a component of that type, the composite type is treated as a limited type until the earliest place within the immediate scope of the declaration of the composite type and after the full declaration of the limited private type.
- Implementation Guideline:* Use a composite type declared in a nested package similar to the example in S1 above.
- T10. Check that an equality operation may be declared for a limited private type (see RM 6.7/4) inside the package that declares it, and that an equality operation may be declared for a limited private type outside the package as well (see IG 7.4.2/T10).

Chapter 8

Visibility Rules

8.1 Declarative Regions

Semantic Ramifications

S1. Consider the following package declaration:

```
package Q is
  package P is
    P : INTEGER;
  end P;
end Q;
```

The declaration of object P occurs immediately within the innermost declarative region (the region created for package P). Package P's declaration itself occurs immediately within the declarative region defined for Q. The first and third occurrences of P are considered to occur in Q's declarative region (RM 8.1/7). If Q is a library package, it is considered to occur within STANDARD's declarative region (RM 8.6/2).

S2. The rules prohibiting homographs from being declared in the same declarative region (RM 8.3/17) can be used to check that an implementation supports the correct definition of a declarative region.

Changes from July 1982

S3. Subunits of subunits are contained in the declarative region of the parent unit.

Changes from July 1980

S4. The term "declarative region" is introduced.

Test Objectives and Design Guidelines

T1. Check that declarative regions are defined correctly (see IG 8.3/T1-T8).

8.2 Scope of Declarations

Semantic Ramifications

S1. Ada distinguishes the *scope* of a declaration from its *visibility*. At a given point in Ada text, an identifier may have potential associations with several declarations (i.e., several possible meanings). Which of these meanings is chosen is determined (for nonoverloaded identifiers) by the *visibility* rules. In essence, *scope* rules determine the region of text in which a declared entity can potentially be referenced, and *visibility* rules (for nonoverloaded identifiers) determine which potentially referenceable declaration is actually associated with the occurrence of a name. (In some other languages, the term "scope" or "name scope" is used in the sense of "visibility" here.) For overloaded identifiers, operator symbols, or character literals, the visibility rules and the overloading resolution rules together determine which declaration is meant.

S2. The difference between the visibility of a name and its scope is illustrated by this example:

```
procedure P is
  I : INTEGER;
```

```

package Q is
    I : CHARACTER;
end Q;
begin
    I := 3;
    Q.I := 'C';
end P;

```

The scope of Q.I and P.I both include the assignment statements, but the visibility rules determine that the I in the first assignment statement unambiguously refers to the integer variable declared in P, whereas the I in the second assignment statement refers to the variable declared in package Q. Because Q.I's scope extends outside package Q, the variable can be referenced outside package Q. (See IG 8.3/S for further discussion.)

s3. Scope rules for library units are not mentioned explicitly in RM 8.2 since library units follow the usual rules, given that library units are considered to be declared implicitly and immediately within the predefined package STANDARD (RM 8.6/2). For example, consider the following library unit:

```

package P is
    type NEW_CHAR is ('A', 'B', 'C');
    type NSTR is array (POSITIVE range <>) of NEW_CHAR;
    X : STRING (1..1) := (1 => ASCII.NUL);
end P;

```

The references to POSITIVE, STRING, and ASCII, which are declared in STANDARD, are legal because package P is considered to be declared in STANDARD's declarative region, and hence, the scopes of the declarations of POSITIVE, STRING, ASCII, and ASCII.NUL include the declarations of NSTR and X. In particular, because the scope of the declaration of NUL in package ASCII includes X's declaration, the name ASCII.NUL is legal. Now suppose we write:

```

with P;
package Q is
    Y : STRING (1..3) := STRING("ABC");      -- illegal
    Z : P.NSTR (1..3) := "CCC";              -- ok
end Q;

```

RM 8.2/1 mentions that a declaration can associate some notation with a declared entity. This phrase refers to the implicit declaration of basic operations and the notation associated with these operations. In particular, string literal formation and assignment are basic operations (RM 3.3.3/4 and RM 3.3.3/7). The scope of the notation associated with these operations for P.NSTR is the same as P.NSTR's scope (RM 8.3/18). Because of the with clause, RM 8.6/2 requires that package P appear before package Q in STANDARD's declarative region, so the scope of NSTR's declaration includes package Q (RM 8.2/3). The type of "ABC" can, therefore, be either STRING or NSTR, since this string literal is within the scope of the string literal formation operations declared for STRING and NSTR. The type of the operand of a conversion must be determined independently of the context (RM 4.6/3). Since the type of "ABC" is ambiguous, the conversion is illegal. On the other hand, the declaration of Z is legal since P.NSTR's assignment operation is visible, the overloading resolution rules determine that the string literal must have type P.NSTR (RM 8.7/8), and P.NSTR's string literal formation operation is visible.

s4. See IG 8.6/S for further discussion of the treatment of library units and the scope of declarations.

Changes from July 1982

S5. There are no significant changes.

Changes from July 1980

S6. The scope of library units is now explained by the notion that library units are implicitly declared in STANDARD, so no additional rules are needed here.

Test Objectives and Design Guidelines

The scope of a declaration can only be checked by seeing if it is visible, directly or by selection. The visibility checks are performed in IG 8.3/T21-T29.

T1. Check that the scope of declaration in a declarative region extends from the beginning of the declaration to the end of the declarative region. In particular, check that the scope does not start from the beginning of the declarative region (see IG 8.3/T21-T29).

Implementation Guideline: This is checked by seeing whether homographs implicitly declared earlier in the same declarative part or in an outer declarative region are visible before the explicit declaration occurs.

T2. Check that the scope of a declaration occurring immediately within the visible part of a package declaration extends outside the package declaration (see IG 8.3/T51).

Implementation Guideline: Check that declarations occurring in an inner package are not visible, nor are the formal parameters of a subprogram or generic unit.

Check that the scope of a declaration occurring in a package nested within the visible part of a package declaration extends outside the package declaration (see IG 8.3/T51).

T3. Check that the scope of an entry declaration extends outside a task declaration or a task type declaration (see IG 8.3/T52).

T4. Check that the scope of a record component declaration (including a discriminant component) extends outside the record's type (see IG 8.3/T53).

T5. Check that the scope of a formal parameter of a subprogram or an entry extends outside the subprogram's or entry's specification (see IG 8.3/T54).

Implementation Guideline: Include subprograms declared by explicit declarations (see IG 6.4/T3), by renaming declarations (see IG 8.5/T13), by generic instantiations (see IG 8.3.e/T3), by derivation (see IG 3.4/T18), and by the occurrence of a subprogram body (see IG 6.4/T3).

T6. Check that the scope of a generic formal parameter extends outside the generic formal part (see IG 8.3/T55).

8.3 Visibility of Identifiers**Semantic Ramifications**

S1. Overloading resolution can affect what declarations are visible. For example:

```

type R is
  record
    X : INTEGER;
  end record;

X : INTEGER := 1;

A : R := (X => 1);           -- X is visible by selection
B : STRING (1..1) := (X => 1); -- X is directly visible

```

The overloading resolution rules determine that the first aggregate has type R. Since R is a record type, the X in the aggregate denotes the record component name (which is visible by selection; RM 8.3/9). In the declaration of B, the overloading resolution rules imply that the aggregate has type STRING, so the X must be an expression denoting an index. In this case, record component X is not visible; X denotes the directly visible declaration of variable X.

S2. If X in the declaration of B is a function call, overloading resolution interacts with the visibility rules in a more complex fashion:

```

type R is
  record
    X : INTEGER;
  end record;

function X return INTEGER is          -- X1
begin ... end X;

function X return FLOAT is           -- X2
begin ... end X;

package P is
  A : R := (X => 1);                  -- X is visible by selection
  B : STRING (1..1) := (X => 1);      -- X1 and X2 are directly visible
end P;

```

Given that the aggregate in B's initialization expression is an array aggregate (a fact determined by the overloading resolution rules), both functions X are directly visible. Since B's index subtype is a subtype of INTEGER, the overloading resolution rules determine that X1 is to be called. So the overloading resolution rules can help to determine both what is visible and which of the visible declarations is meant.

S3. Since a name cannot be referenced within its own declaration, the name of a task type cannot be used in its own specification:

```

task type T is
  entry E (X : T);                  -- illegal
end T;

```

(The name T can be used in the task body, however, since a task body is not a declaration (RM 3.9/2). Use of the name within T's body is limited, however. The name cannot be used as a type mark because T is the name of the enclosing task body; see RM 9.1/4.)

S4. Similarly, a package declared by a generic instantiation is not directly visible within its own declaration (RM 8.3/5), e.g.:

```

package P is new Q(P.A);           -- illegal

```

The name P.A is illegal because the context of P's use requires a directly visible declaration of P and no P is directly visible where P.A occurs. To see this, first note that a declaration can be directly visible if:

- it occurs (explicitly or implicitly) earlier in the same declarative region;
- it occurs (explicitly or implicitly) earlier in an enclosing declarative region;
- a use clause makes it directly visible.

Now consider: RM 8.3/5 specifies that the instance, P, is not visible until after the end of the

instantiation, so the prefix P in P.A cannot denote the package declared by the instantiation. Since package P is not overloadable, any implicitly declared P that occurs (earlier) in P's declarative region is hidden (RM 8.3/17); moreover, any P declared in an outer declarative region is hidden from the beginning of the instantiation (RM 8.3/15). Furthermore, the prefix P cannot denote a declaration made visible by a use clause, because no homograph of P can be made visible by a use clause within the immediate scope of package P (RM 8.4/5). Since these are the only ways a declaration can be directly visible, there is no directly visible P.

S5. A package declared by a generic instantiation is also not visible by selection within its own declaration. For example, suppose P is declared as a library unit:

```
package P is new Q(STANDARD.P.A); -- illegal
```

Even though P is declared in STANDARD, its declaration cannot be referred to within itself.

S6. It is possible to use the identifier P within an instantiation of package P, but only if it refers to a declaration that is visible by selection. For example:

```
procedure S is
```

```
  type R is
    record
      P : INTEGER := 3;
    end record;
```

```
  X : R;
```

```
  package PACK is
    P : INTEGER := 3;
  end PACK;
```

```
  function F (P : INTEGER) return INTEGER is ... end F;
```

```
  generic
    I1 : INTEGER;
    I2 : INTEGER;
    R1 : R;
    I3 : INTEGER;
    P : INTEGER;
```

```
  package P is
  end P;
```

```
  package NESTED is
    package P is
      new S.P (X.P, PACK.P, (P => 1), F(P => 3), P => 3);
    end NESTED;
```

```
  end S;
```

All the uses of P in package P's instantiation are legal because each declaration denoted by each P is visible by selection (RM 8.3/7-13).

S7. None of the above uses of P are legal, however, if the instantiation declares a subprogram:

```
generic
  I1 : INTEGER;
  I2 : INTEGER;
```

```

R1 : R;
I3 : INTEGER;
P  : INTEGER;
procedure P;

package NESTED is
  procedure P is
    new S.P (X.P, PACK.P, (P => 1), F(P => 3), P => 3);
    -- illegal uses of P
  end NESTED;

```

RM 8.3/16 says that within a generic instantiation that declares a subprogram, a declaration is visible neither by selection nor directly if it has the same designator as the instantiated subprogram. Each of the uses of P violate this rule.

S8. Since a subprogram can be instantiated as an operator symbol, the corresponding operator cannot be named within the instantiation:

```

package S is
  procedure "+" is new R(3 + 4, STANDARD."+"(3, 4)); -- illegal
end S;

```

The uses of operator + are illegal (since no "+" is directly visible (as an infix operator) within the instantiation, and no "+" operation is visible by selection either).

S9. The restrictions of RM 8.3/16 only apply within subprogram specifications and instantiations. This paragraph does not apply to the default name of a generic formal subprogram, e.g.:

```

generic
  with function "+" (L, R : INTEGER) return INTEGER
  is "+";
  with function "-" (L, R : INTEGER) return INTEGER
  is STANDARD."-";
procedure P;
-- illegal
-- ok

```

The first default name is illegal because of RM 8.3/5; the only "+" having the correct parameter and result type profile is the formal parameter itself, and this entity cannot be referenced within its own declaration. The second default name is legal because it does not denote the formal parameter being declared.

S10. RM 8.3/6 says visibility is either by selection or direct. This means that these are the only forms of visibility; moreover, once a given occurrence of an identifier has been associated with a declaration, it is either associated by selection or directly (RM 8.3/14).

S11. Hiding of an identifier occurs as soon as a new declaration of the identifier is encountered, e.g.:

```

procedure R is
  I : constant INTEGER := 4;
  type R is record
    J : INTEGER := I;      -- R.I would be illegal
    K : FLOAT digits I;    -- R.I also illegal
  end record;

```

S12. It is sometimes the case that a declaration can only be made visible by selection (e.g., the component name in a record). It is also possible for a declaration to be visible only directly and not by selection, e.g.,

```

declare
  X : INTEGER;
begin
  ...
end;

```

Within the block, X is only visible directly since the block does not have a name, so X cannot be named by selection.

S13. Finally, it is possible for a declaration to be visible both directly and by selection, although any given occurrence of an identifier will be associated with the declaration by only one of these methods. For example, a declaration in the visible part of a package specification can be named directly within the package or by selection in an expanded name whose prefix denotes the package.

S14. The selected component form of naming can only be used to access identifiers whose declarations are in scope. Outside of a procedure, selected component notation cannot be used to access any entities declared inside the procedure, including formal parameter names. e.g.:

```

package body R is
  I : FLOAT;
  function R (I : INTEGER) return INTEGER is
    J : INTEGER;
  begin
    ...
  end R;
  function J return INTEGER is begin ... end J;
begin
  -- references to R.I and R.J are illegal

```

The cited references are illegal because the scopes of INTEGER I and J do not extend outside of function R. Since the declaration of function R hides the package R declaration, references to R.I and R.J can only be attempts to reference components of function R. Since no such components are in scope, the references are illegal.

S15. If "package body" is replaced by "procedure", the declaration of function R does not hide the enclosing procedure declaration. Since procedure R is referenceable in this case, the R in R.I is ambiguous. RM 4.1.3/18 states that R.X is therefore illegal for all X.

S16. Components of private types cannot be accessed using selected component notation except within the private part of a package and within the corresponding package body, since the scope of such components extends only over the private part and the package body.

S17. Use clauses can make homographs directly visible:

```

package Q is
  procedure P (X : INTEGER);
end Q;

package R is
  procedure P (Y : INTEGER);
end R;

use Q, R;

```



```

P(3);           -- illegal; ambiguous
P(X => 3);       -- legal; Q.P is called

```

The use clause makes both Ps visible even though they are homographs (RM 8.4/6), so the first call is illegal. The second call is legal, since formal parameter names can be used to resolve overloaded calls (RM 6.6/3). (Parameter names are ignored when deciding whether two subprograms are homographs (RM 8.3/15 and RM 6.6/1, /6).)

S18. A declaration in an outer declarative region can be hidden by a declaration in an inner region (RM 8.3/15), and an implicit declaration can be hidden by an explicit declaration declared in the same region (RM 8.3/17). A hidden declaration is not directly visible, but the inverse is not true; a declaration can be both "not hidden" and not "directly visible." For example, a declaration in a package need not be directly visible even though it is not hidden:

```

package P is
  X : INTEGER;
end P;

A : INTEGER := X;           -- illegal

```

Within the initialization expression for A, P.X is not hidden (it is not in the same or an outer declarative region), but the declaration of X is also not directly visible.

S19. RM 8.3/17 speaks of an explicit declaration hiding an earlier implicit declaration. This can occur as follows:

```

package P is
  type T is range 1..10;
  -- implicit declaration of "+" and other operations
  X : T := 3 + 4;           -- implicitly declared +
  function "+" (L, R : T) return T;
  type REC is
    record
      C : T := 3 + 4;       -- explicitly declared +
    end record;
end P;
use P;

Y : T := 3 + 4;           -- explicitly declared +

```

The explicitly declared "+" hides the implicitly declared "+" throughout the scope of the explicit declaration. This scope includes the use of "+" in the declaration of REC, and since "+" is declared in the visible part of a package, the scope of "+" extends outside the package (RM 8.2/3) and includes the declaration of Y. The scope of the implicitly declared "+" also extends outside package P, but RM 8.3/17 ensures that only the explicitly declared "+" is visible, either directly or by selection.

S20. It is also possible for an explicit declaration to hide a later implicit declaration:

```

package P is
  type T is private;
  function "+" (L, R : T) return T;
private
  type T is range 1..10;
  -- implicit declaration of "+"

```

The implicit declaration of "+" is hidden by the earlier explicit declaration. Similar effects can

occur when the full declaration is a derived type. In particular, it is possible for two implicit declarations of homographs to occur. When one of the two is the declaration of an enumeration literal and the other is the declaration of a derived subprogram, the declaration of the derived subprogram hides the enumeration literal (see IG 8.3.d/S). It is also possible for an enumeration literal, derived subprogram, and statement label, block name, or loop name to be declared implicitly in the same declarative region (see IG 8.3.a/S). In this case, the enumeration literal and derived subprogram are hidden.

S21. A derived type declaration can derive two or more derived subprogram homographs (AI-00012). One way to derive homographs is:

```
package R is
  type T is private;

  package P is
    type U is range 1..10;
    procedure Q (X : T; Y : U);
    procedure Q (X : U; Y : T);
    procedure Q (X : U; Y : U);
  end P;
private
  type T is new P.U;      -- legal; AI-00012
  -- three implicit declarations of Q (X : T; Y : T);
```

The three implicitly declared derived subprograms are homographs of each other.

S22. Another way to derive homographs is when the parent type has two or more derivable subprograms that are homographs:

```
generic
  type A is (<>);
  type B is private;
package G is
  function NEXT (X : A) return A;
  function NEXT (X : B) return B;
end;

package P is new G (A => BOOLEAN, B => BOOLEAN);
```

The two declarations of NEXT that occur in instance P are homographs, but the instantiation is legal since the actual parameters match the formal parameters (see RM 12.3/22 and AI-00012). This sort of construction provides another way of deriving subprogram homographs:

```
generic
  type A is (<>);
  type B is private;
package G is
  type T is private;
  function NEXT (X : A; Y : T) return A;
  function NEXT (X : B; Y : T) return B;
end;

package P is new G (A => BOOLEAN, B => BOOLEAN);

type NT is new P.T;      -- legal; AI-00012
```

The derived type declaration is allowed even though both NEXT subprograms will be derived, and the subprograms will have the same parameter and result type profiles (AI-00012).

S23. RM 8.3/18 says the notation associated with a basic operation is visible throughout the scope of the operation, but consider the use of a type name in a conversion or qualified expression:

```
package P is
  type INT is range 1..10;
end P;

-- INT is not directly visible here

X : P.INT := INT(5);           -- illegal
Y : P.INT := INT' (5);         -- illegal
```

The conversion and qualified expression are illegal because the identifier INT must be associated with a declaration by the visibility rules, and no INT is directly visible. Although INT is not directly visible, the conversion operation is visible and can be used. For example:

```
subtype PINT is P.INT;
Z1 : P.INT := 5;               -- implicit conversion
Z2 : P.INT := PINT' (5);       -- explicit conversion
```

The explicit conversion using PINT is legal because PINT is directly visible and because the conversion operation associated with PINT's base type is visible.

S24. Special interactions between scope and visibility rules are treated in subsections addressing:

- a. Labels
- b. Loop Parameters
- c. Records
- d. Enumeration Literals
- e. Parameters
- f. Packages

Interactions with separate compilation are treated in IG 8.4. Interactions with overloading are treated in IG 6.6 and IG 6.7.

Changes from July 1982

S25. The visibility rules don't apply to an identifier given as a pragma argument, e.g., `pragma OPTIMIZE (TIME)` is legal even though no declaration of TIME is directly visible.

S26. Hiding rules for subprogram names are explicitly extended to include subprograms declared for operator symbols.

S27. The designator declared by a generic instantiation is not visible either directly or by selection until after the instantiation.

Changes from July 1980

S28. The visibility rules do not apply to pragmas, the identifier of a pragma argument (e.g., ON in `pragma SUPPRESS (RANGE_CHECK, ON => P)`), reserved words, or attribute designators.

S29. The term "homograph" is introduced.

Legality Rules

- L1. If the visibility rules allow an identifier to be associated with more than one declaration, then the overloading rules must allow exactly one of these declarations as the meaning of the identifier (RM 8.3/4).
- L2. The identifier or operator symbol being declared by a declaration is not visible within its own declaration (except for an identifier declared in a package specification as the name of the package) (RM 8.3/5).
- L3. A declaration is not visible, either directly or by selection, outside its scope (RM 8.3/6-14).
- L4. Within the scope of a homograph of a declaration given in an outer declarative region, the outer declaration is not directly visible (RM 8.3/15).
- L5. Within the specification of a subprogram declared by a subprogram declaration, a subprogram body, a generic instantiation, or a renaming declaration, every declaration with the same designator as the subprogram is visible neither by selection nor directly (RM 8.3/16).
- L6. Within the declaration of an entry or the formal part of an accept statement, every declaration with the same identifier as the entry is visible neither by selection nor directly (RM 8.3/16).
- L7. Two declarations that occur immediately within the same declarative region must not be homographs, unless either or both of the following requirements are met: (a) exactly one of them is the implicit declaration of a predefined operation; (b) one or more of them is the implicit declaration of a derived subprogram (RM 8.3/17 and AI-00012).
- L8. An operator is directly visible if and only if the corresponding operator declaration is directly visible (RM 8.3/18).

Test Objectives and Design Guidelines

Tests T1-T8 check that pairs of homographs are illegal if they occur within each of the various declarative regions. Tests T11 and T12 check visibility within a declaration. Tests T21-T29 checking the hiding of outer declarations, and T30-T33 check other kinds of hiding. T41 checks visibility of operations declared in a package. Tests T51-T56 check the various forms of selective visibility.

T1. Check that the formal parameters of a nongeneric subprogram specification and the declarative part of the subprogram's body form a single declarative region. In particular, check that:

- two formal parameters cannot have the same identifier (see IG 8.3.e/T1).
- a name declared explicitly in the body's declarative part cannot be the same as the name of a formal parameter (see IG 8.3.e/T1).

Check that two homographs cannot be declared explicitly in the declarative part of the subprogram's body.

Implementation Guideline: In particular, check a sampling of the following forms of homographs: two subprograms explicitly declared by combinations of the following methods: a subprogram declaration, a renaming declaration, an enumeration literal specification (see AI-00330), and a generic instantiation; subprograms vs. nonoverloadable declarations (IG 6.6/T1 checks subprogram homographs and subprogram declarations vs. nonoverloadable declarations); and two nonoverloadable declarations. The nonoverloadable declarations to be used are those for variables, constants, named constants, exceptions, types, subtypes, packages, task units, and generic units (packages, functions, and procedures). (Note: checks for duplicate

block, loop, and statement names are performed in IG 8.3.a/T1. IG 8.3.a/T7 checks that block, loop, and statement names cannot be the same as other kinds of names declared in the same declarative region.)

- T2. Check that the visible and private parts of a package specification form a single declarative region shared with the declarative part of the package body. In particular, check that two homographs cannot be declared within the following parts of the declarative region: visible-visible, visible-private, visible-body, private-private, private-body, and body-body (see IG 8.3.f/T2).

Implementation Guideline: Use the guideline given for T1.

Implementation Guideline: Include a check when the package body is declared as a subunit and the other declarations occur in the package specification.

- T3. Check that a task specification and the declarative part of a task's body form a single declarative region. In particular, check that:

- single entry homographs cannot be declared in a task specification (see IG 9.5/T94).
- a single entry and entry family cannot have the same identifier (see IG 9.5/T94).
- a procedure declared explicitly in the declarative part of the task's body cannot be a homograph of an entry declared in the task specification (see IG 9.5/T94).

Implementation Guideline: Include procedures declared by a subprogram declaration, renaming declaration, and generic instantiation. Include a check when the task body is declared as a subunit.

- an explicit declaration in the declarative part of the task's body cannot have the same identifier as that of an entry family.

Implementation Guideline: Include all forms of overloadable and nonoverloadable declaration in the body (see guideline for T1).

Implementation Guideline: Include a check when the task body is declared as a subunit.

- two explicit declarations appearing in the task body's declarative part cannot be homographs.

Implementation Guideline: Include the following pairs of declarations: two subprogram homographs, an overloadable vs. a nonoverloadable declaration, two nonoverloadable declarations (see the guideline for T1).

- T4. Check that a generic formal part, a generic package specification, and a generic package body form a single declarative region. In particular, check that:

- homographs are not allowed within a generic formal part (see IG 12.1/T9).

Implementation Guideline: Include the following pairs of declarations: two formal subprogram homographs; a formal subprogram and a nonoverloadable formal parameter having the same identifier; and two nonoverloadable formal parameters having the same identifier. The nonoverloadable forms of formal parameter declarations are a formal object declaration and a formal type declaration (discrete, integer, float, fixed, array, access, and private).

- an explicit declaration in the package specification cannot declare a homograph of a name declared in the generic formal part (see IG 12.1/T9).

Implementation Guideline: See the above guideline.

- an explicit declaration in the body of a generic package cannot declare a homograph of a name declared in the generic formal part (see IG 12.1/T9).

Implementation Guideline: Include cases where the package body is given as a subunit.

In addition, repeat the checks for nongeneric package declarations as described in T2.

- T5. Check that the generic formal part, the formal parameters, and the declarative part of a generic subprogram's body form a single declarative region. In particular, check that:

- homographs are not allowed within the generic formal part (see IG 12.1/T9).
Implementation Guideline: See the guideline for T4.
- the identifier of a generic formal parameter cannot be the same as the identifier of a formal parameter of the generic subprogram (see IG 12.1/T9).
Implementation Guideline: Check for each form of generic formal parameter.
- an explicit declaration in the body of a generic subprogram cannot declare a homograph of a name declared in the generic formal part (see IG 12.1/T9).
Implementation Guideline: Include cases where the subprogram body is given as a subunit.

In addition, repeat the checks for nongeneric subprogram declarations as described in T1.

- T6. Check that a record type declaration forms a single declarative region. In particular, check that two discriminants cannot have the same identifier, two nondiscriminant components cannot have the same identifier (even if they appear in different variant parts), and a discriminant and a nondiscriminant component cannot have the same identifier (see IG 8.3.c/T1).

Check that the discriminants in a private or incomplete type declaration cannot have the same identifier.

Check that the discriminants in a generic formal type declaration cannot have the same identifier.

- T7. Check that the formal parameters of a subprogram declared by a renaming declaration form a declarative region, and hence, cannot have the same identifier (see IG 8.3.e/T1).

Check that formal parameters of an entry declaration cannot have the same identifier (see IG 8.3.e/T11).

Check that a formal parameter of a subprogram declared by a renaming declaration can be the same as a name declared in the renamed subprogram's body.

- T8. Check that two homographs cannot be declared explicitly in the declarative part of a block.

Implementation Guideline: See the guideline for T1.

- T9. Check that a derived type declaration is allowed if it derives two or more subprogram homographs.

Implementation Guideline: Check the following cases:

- the subprograms derived from the parent type become homographs because of the substitution of the derived type for the parent type;
- the parent type is declared in a generic instance and has two or more derivable subprograms that are homographs.

The derived type declaration should be given in a package specification (in the visible and the private part), a package body, a subprogram body, and a block. For packages and subprograms, repeat for generic units.

- T11. Check that a declared entity is not visible (either directly or by selection) within its own declaration, except for package declarations.

Implementation Guideline: Check within the following forms of declaration: object declaration (both constant and variable), type declaration (for each kind of type: enumeration, integer, float, fixed, array, record, access, and task), subtype declaration, subprogram declaration, renaming declaration, number declaration, formal parameter declaration, generic formal parameter declaration (object, type, and subprogram, e.g., the default name), or generic subprogram declaration.

- T12. Check that within a subprogram specification (appearing as a subprogram declaration, renaming declaration, generic formal subprogram parameter), single entry, entry family, or subprogram instantiation, no declaration of the designator of the declared subprogram or entry is visible, either directly or by selection (see IG 6.1/T12, where some of this objective is performed).

Implementation Guideline: Use all possible forms of visibility by selection: a visible part of a package, a record component, a record aggregate, a named parameter association in a function call, a named generic association, and an expanded name whose prefix denotes an enclosing construct.

Implementation Guideline: Check both identifiers and operator symbols. For operator symbols, use the operator as an infix or prefix operator as well as in function notation.

Check that in the instantiation of a package, a declaration having the package identifier is visible by selection.

- T13. Check that subprograms or single entries having the same identifier can be declared in the same declarative region if they do not have the same parameter and result type profile (see IG 6.6/T2 and IG 9.5/T95).

- T21. Hiding in packages: check that a declaration given in an outer declarative region is hidden by the declaration of a homograph occurring in an inner declarative region associated with a package. Within the package's visible part, private part, and body, check that the outer declaration is directly visible prior to the declaration of the inner homograph (see IG 8.3/T1).

Check that the hidden declaration is visible by selection (except within the formal part of a subprogram or entry specification having the same designator as the hidden declaration; see IG 8.3/T12 for this special case).

Implementation Guideline: Check the following pairs of homographs (each member of the pair should be declared in an outer declarative region):

- subprogram vs. subprogram
Implementation Guideline: Include subprograms declared by renaming declarations and generic instantiations in this and subsequent checks that involve subprograms.
- single entry vs. subprogram
- enumeration literal vs. function
- subprogram, generic formal subprogram, or single entry vs. a nonoverloadable declaration (variable, constant, exception, type, subtype, package, task unit, generic unit, entry family, formal parameter of a subprogram, entry, or generic subprogram), generic formal object or type parameter, a block name, loop name, or statement label; block names, loop names, and statement labels are checked in IG 8.3.a/T9)
- nonoverloadable declaration vs. nonoverloadable declaration

Implementation Guideline: Include cases in which the package body is given as a subunit.

- T22. Hiding in subprograms: check that a declaration given in an outer declarative region is hidden by the declaration of a homograph occurring in an inner declarative region associated with a subprogram. Within the subprogram's formal part and body, check that the outer declaration is directly visible prior to the declaration of the inner homograph.

Implementation Guideline: Use subprograms declared by a subprogram declaration and body, a subprogram body only, a renaming declaration, a formal generic subprogram parameter declaration, and a generic instantiation. In the case of renaming declarations, generic formal parameters, and generic instantiations, only check that the formal parameters hide outer declarations having the same identifier.

Implementation Guideline: Use some examples like the following in which a declaration of a formal parameter hides a declaration given in an outer declarative region even though the outer declaration occurs after the subprogram specification.

```
procedure OUTER is
  function F is new GF (INTEGER);    -- F1 returns INTEGER
```

```

procedure INNER (
  X : INTEGER := F;           -- calls F1
  F : FLOAT);                 -- hides F1

function F is new GF (FLOAT); -- F4 returns FLOAT

procedure INNER (
  X : INTEGER := F;           -- illegal; write OUTER.F
  F : FLOAT) is
begin
  X := INTEGER(F);            -- formal parameter
end INNER;

```

Check that the hidden declaration is visible by selection (except within the formal part of a subprogram or entry specification having the same designator as the hidden declaration; see IG 8.3/T12 for this special case).

Implementation Guideline: Check the pairs of homographs given in the guideline for T21.

Implementation Guideline: Include cases when the subprogram body is given as a subunit.

- T23. Hiding in tasks and task type declarations: check that a declaration given in an outer declarative region is hidden by the declaration of a homograph occurring in an inner declarative region associated with a task or task type declaration. Within the task specification and body, check that the outer declaration is directly visible prior to the declaration of the inner homograph.

Implementation Guideline: Use some examples like the following in which an entry declaration appearing in a task specification hides a declaration given in an outer declarative region even though the outer declaration occurs after the package specification:

```

generic
  type T is private;
package GP is
  procedure PROC (Y : T);
end GP;

package body GP is
  ...
end GP;

procedure OUTER is
  procedure P is new GP (INTEGER); -- P1

  task INNER is
    entry P (X : INTEGER);
    entry P (X : FLOAT);
  end INNER;

  procedure P is new GP (FLOAT); -- P2

  task body INNER is
  begin
    P (Y => 1); -- illegal: P1 hidden
    P (Y => 1.0); -- illegal: P2 hidden
    declare
      procedure P is new GP (FLOAT); -- P3
    begin
      accept P (X : FLOAT) do
        P; -- deadlock; calls entry
      end P;
    end;
  end;

```



```

      P (1.0);      -- calls P3
    end;
  end INNER;

```

Check that the hidden declaration is visible by selection (except within the formal part of a subprogram or entry specification having the same designator as the hidden declaration; see IG 8.3/T12 for this special case).

Implementation Guideline: Check the pairs of homographs given in the guideline for T21.

Implementation Guideline: Include cases in which the task body is given as a subunit.

- T24. Hiding in generic packages:** check that a declaration given in an outer declarative region is hidden by the declaration of a homograph occurring in an inner declarative region associated with a generic package. Within the package's generic formal part, visible part, private part, and body, check that the outer declaration is directly visible prior to the declaration of the inner homograph.

Implementation Guideline: Use some examples like those given for T21, including cases where the inner declaration occurs in the generic formal part.

Check that the hidden declaration is visible by selection (except within the formal part of a subprogram or entry specification having the same designator as the hidden declaration; see IG 8.3/T12 for this special case).

Implementation Guideline: Check the homographs described in the guideline for T21.

Implementation Guideline: Include cases in which the package body is given as a subunit.

- T25. Hiding in generic subprograms:** check that a declaration given in an outer declarative region is hidden by the declaration of a homograph occurring in an inner declarative region associated with a generic subprogram. Within the subprogram's generic formal part, formal part, and body, check that the outer declaration is directly visible prior to the declaration of the inner homograph.

Check that the hidden declaration is visible by selection (except within the formal part of a subprogram or entry specification having the same designator as the hidden declaration; see IG 8.3/T12 for this special case).

Implementation Guideline: Follow the guidelines for T22, extending the cases to include declarations in the generic formal part (as for T24).

Implementation Guideline: Include cases in which the subprogram body is given as a subunit.

- T26. Hiding in entry declarations:** check that a declaration given in an outer declarative region is hidden by the declaration of a homograph occurring in an inner declarative region associated with the formal part of an entry declaration and the corresponding accept statements. Within the entry's formal part, check that the outer declaration is directly visible prior to the declaration of the inner homograph.

Check that the hidden declaration is visible by selection within the formal part and accept statement (except within the formal part of a subprogram or entry specification having the same designator as the hidden declaration; see IG 8.3/T12 for this special case).

Implementation Guideline: Follow the guidelines for T22.

- T27. Hiding in record type declarations:** check that a declaration given in an outer declarative region is hidden by the declaration of a homograph occurring in an inner declarative region associated with the declaration of a record type. Within the record type's declaration, check that the outer declaration is directly visible prior to the declaration of the inner homograph.

Implementation Guideline: Include a check within a discriminant part.

Check that outer declarations are appropriately hidden within the discriminant part of a private type declaration, incomplete type declaration, or generic formal type declaration.

Check that the hidden declaration is visible by selection.

- T28. Hiding in block statements: check that a declaration given in an outer declarative region is hidden by the declaration of a homograph occurring in an inner declarative region associated with the declarative part of a block statement. Within the declarative part, check that the outer declaration is directly visible prior to the declaration of the inner homograph.

Check that the hidden declaration is visible by selection (except within the formal part of a subprogram or entry specification having the same designator as the hidden declaration; see IG 8.3/T12 for this special case).

- T29. Hiding by a loop parameter: check that a declaration given in an outer declarative region is hidden by the declaration of a loop parameter.

Check that the hidden declaration is visible by selection (except within the formal part of a subprogram or entry specification having the same designator as the hidden declaration; see IG 8.3/T12 for this special case).

- T30. Check that within a generic formal part, no outer declaration having the same identifier as the generic unit is directly visible (see IG 12.1/T5).

Implementation Guideline: Check for both generic subprograms and generic packages.

Implementation Guideline: Check for declarations that are potentially visible because of use clauses as well as for declarations given in an outer declarative region.

Check that within a generic subprogram body, no subprogram or generic subprogram declared in an outer declarative region is hidden (unless the subprogram is a homograph of the generic subprogram). Similarly, check that subprograms can be made directly visible because of a use clause that appears outside the generic subprogram body.

- T31. Check that an implicit declaration of a predefined operator or enumeration literal is hidden by an explicit declaration of a homograph of the operator or literal.

Implementation Guideline: Check each form of explicit declaration: subprogram declaration, renaming declaration, generic instantiation, and (for operators only) generic formal subprogram declaration.

Implementation Guideline: An enumeration literal should also be hidden by a nonoverloadable declaration.

Implementation Guideline: Check that the implicit declaration can occur before or after the explicit declaration, and that in the case where the explicit declaration comes last, the implicit declaration is visible prior to the occurrence of the explicit declaration. In particular, if the implicit declaration occurs in the visible part of a package and the explicit declaration occurs in the private part, then the implicit declaration should be visible outside the package. On the other hand, if the explicit declaration also occurs in the visible part of the package, only the explicit declaration is visible outside the package.

Implementation Guideline: The homographs for an operator should have different formal parameter names, and where the implicit declaration is hidden, it should be impossible to invoke it even when named associations are used.

- T32. Check that an implicit declaration of a predefined operator or enumeration literal is hidden by a derived subprogram homograph.

Implementation Guideline: The homographs for an operator should have different formal parameter names, and where the implicit declaration is hidden, it should be impossible to invoke it even when named associations are used.

- T33. Check that an implicit declaration of a block name, a loop name, or a statement label hides the declaration of an enumeration literal or a derived subprogram declared by a derived type definition.

Implementation Guideline: Include cases where an enumeration literal, derived subprogram, and block name, loop name, or statement label all have the same identifier.

- T41. Check that basic operations (except for explicit type conversion and type qualification; see below) are visible throughout their scope, and, in particular, are visible even when the type that caused the operations to be declared is declared in the visible part of a package and no use clause is applied to the package.

Implementation Guideline: The basic operations to be checked are: implicit type conversion, assignment, allocators, membership tests, short-circuit control forms, selected components, indexed components, slices, numeric literals, the literal null, a string literal, an aggregate, and attributes. Check for each class of type: integer, enumeration, floating point, fixed point, array, record, access, and private.

Check that the usual visibility rules apply to the type mark occurring in an explicit type conversion or a qualified expression. In particular, if the type mark is the simple name of a type declared in a type declaration and the type declaration is not directly visible, then the type conversion or qualified expression is illegal.

Check that operator symbols and enumeration literals declared in the visible part of a package are not directly visible outside the package unless a use clause makes them visible.

Implementation Guideline: Check for each class of type: integer, enumeration, floating point, fixed point, array, record, access, and private.

- T51. Check that a declaration given in the visible part of a package is visible by selection from outside the package (see IG 4.1.3/T20-T28).

Check that declarations in the visible part of a package nested within the visible part of a package are visible by selection from outside the outermost package.

- T52. Check that an entry declaration of a task type is visible by selection when the prefix is appropriate for the task type (see IG 9.5/T for various tests).

- T53. Check that a component of a record value (or a discriminant of a value having a private or generic formal type with discriminants) is visible by selection (see IG 4.1.3/T1).

Check that a component of a record type is visible by selection as a choice in an aggregate (see IG 4.3.1/T6).

Implementation Guideline: Include components that are discriminants.

- T54. Check that the formal parameter of a subprogram or an entry is visible by selection in a named association of a corresponding subprogram or entry call (see IG 6.4/T3 and IG 9.5/T81).

- T55. Check that a generic formal parameter is visible by selection in a named generic association of a corresponding generic instantiation (see IG 12.3/T4).

- T56. Check that any declaration that occurs immediately within a declarative region (other than a record type declaration) is visible by selection in an expanded name whose prefix denotes the enclosing construct (except when visibility by selection is not allowed) (see IG 4.1.3/T7).

8.3.a Labels, Loop Names, and Block Names

Semantic Ramifications

- S1. RM 5.1/3 states that label, block, and loop names are implicitly declared within the innermost enclosing block or body that encloses the labeled statement, loop, or block, e.g.:

```

procedure Q is
  L : INTEGER;
begin
  begin                -- beginning of block
    <L> ...             -- legal
  end;
end Q;
```

The label is implicitly declared within the enclosing block, and so does not conflict with the declaration of L in the enclosing procedure. If the inner block were not present, label L would be declared at the end of Q's declarative part. Since variable L and label L are homographs, and since label L is implicitly declared but is not a predefined operation or a derived subprogram, the homographs are not allowed (RM 8.3/17).

S2. RM 5.1/4 requires that label, block, and loop names be unique within the entire body. This means that implementations must treat such names specially. For example:

```

procedure P is
begin
  begin
    <<L>>    ...
  end;

  for I in 1..10 loop
    <<L>>    ...      -- illegal
  end loop;
end P;

```

The second declaration of label L is illegal even though it occurs within a different declarative region, and even though the first label is only implicitly declared within the block and the second label is implicitly declared in P's declarative part.

S3. A label, block name, or loop name can be the same as an implicitly declared enumeration literal or a derived subprogram (RM 8.3/17):

```

package P is
  type ENUM is (E1, E2);
  function F return ENUM;
end P;

procedure Q is
  type NEW_ENUM is new P.ENUM;
  -- implicit declaration of E1, E2, F
  X : NEW_ENUM := E1;           -- ok
  Y : NEW_ENUM := F;           -- ok
  -- implicit declaration of labels E1 and F
begin
  X := E1;                     -- illegal; only label E1 is visible now
  Y := F;                      -- illegal; only label F is visible now
  <<E1>>    ...                 -- legal
  <<F>>    ...                 -- legal
end Q;

```

Labels E1 and F are implicitly declared at the end of Q's declarative part and are homographs of enumeration literal E1 and function F, which are also declared implicitly in Q's declarative part. Since an enumeration literal is a predefined operation, it is hidden by label E1 (RM 8.3/17, rule (a)). Similarly, since function F is an implicitly declared derived subprogram, label F hides the function (RM 8.3/17, rule (b)). Since the labels are declared at the end of Q's declarative part (RM 5.1/3), the enumeration literal and function are not hidden until after the end of the declarative part.

S4. It is possible to construct a more complex example in which an enumeration literal, a derived subprogram, and a statement label having the same identifier are all implicitly declared in the same declarative region:

```

generic
    type T is private;
package GP is
    type NT is new T;
    function F return NT;
end GP;

procedure Q is
    type ENUM is (E, F);

    package PACK is new GP (ENUM);

    type NEW_ENUM is new PACK.NT;
    -- derives enumeration literal F and function F
    X : NEW_ENUM := F;          -- calls function F
begin
    X := F;                    -- illegal
    <<F>> ...
end Q;

```

An enumeration literal is derived for NEW_ENUM because PACK.NT is an enumeration type (see IG 12.3/S) and function F is also derived. Label F is implicitly declared at the end of Q's declarative part. RM 8.3/17 requires us to consider the homographs for F by pairs: (function, enumeration literal), (label, enumeration literal), and (label, function). Derived function F hides enumeration literal F, since the enumeration literal is considered a predefined operation (AI-00002). Similarly, label F also hides the enumeration literal. Finally, derived function F is hidden by label F, but only after the end of the declarative part, since the label is declared at the end of the declarative part (RM 5.1/3).

S5. The occurrence of a label, e.g., <<I>>, should not be confused with the declaration of a label. <<I>> refers to the implicit declaration of I as a label; the context in which <<I>> occurs determines where this implicit declaration occurs. Consider the following example:

```

procedure P is
    I : INTEGER;
    -- implicit declaration of label I
begin
    I := 1;
    <<I>>          -- illegal
end P;

```

The implicit declaration of label I is hidden by the explicit declaration of variable I (RM 8.3/17). The attempt to reference label I after the first statement is illegal, because no label is visible. Similarly:

```

for I in 1..10 loop
    <<I>> null;    -- illegal
end loop;

```

The occurrence of label I is illegal because the only visible declaration of I is the loop parameter's declaration, and a loop parameter is not a label. Label I is implicitly declared in an enclosing declarative part. Normal visibility rules then apply to explicit occurrences of this implicitly declared identifier. In particular, <<I>> is only legal when this implicit declaration of I as a label is visible.

S6. Similarly, the name of a loop cannot be used by an exit statement if it is hidden by the loop parameter declaration or some other intervening declaration:

```

L1:  for L1 in 1..10 loop
      exit L1;           -- illegal
    end L1;              -- ok

```

The name given in the exit statement refers to the loop parameter, and so is illegal. The name given at the end of the loop is, however, legal, since the declarative region associated with a loop statement does not include the end of the statement (just as the end of a package or a subprogram declaration occurs in the containing declarative region). Thus, the last occurrence of L1 refers to the loop name. The loop name can be hidden in other ways as well:

```

L2:  for I in 1..10 loop
      declare
        L2 : INTEGER;
      begin
        exit L2;         -- illegal
      end;
    end loop L2;

```

The occurrence of L2 in the exit statement is illegal because only local variable L2 is directly visible.

S7. Additional consequences of the special rules concerning the declaration of labels, loop names, and block names are discussed in IG 5.1/S.

Legality Rules

- L1. Within the sequence of statements and the (optional) exception handling part of a subprogram body, package body, task body, or generic unit (and excluding within any nested subprograms, packages, tasks, or generic units), no identifiers used for statement labels, block names, or loop names are allowed to be the same (RM 5.1/4).
- L2. For each label used in a goto statement, there must be a corresponding visible implicit label declaration in the innermost block statement, subprogram body, package body, task body, or generic body that encloses the labeled statement (RM 5.9/1-2 and RM 8.3/3).

For each loop name used in an exit statement, there must be a corresponding visible implicit loop name declaration in the innermost block statement, subprogram body, package body, task body, or generic body that encloses the named loop (RM 5.7/3 and RM 8.3/3).
- L3. The identifier of a statement label, a block name, or a loop name must be distinct from any other identifier declared explicitly in the innermost block statement, subprogram body, package body, task body, or generic body that encloses the labeled statement, the named loop statement, or the named block statement (RM 5.1/3 and RM 8.3/17).

Test Objectives and Design Guidelines

- T1. Check that within the body of a subprogram, package, or task (and excluding within nested bodies), a statement label, a block name, or a loop name inside a loop, an accept statement, a block body, or an exception handler cannot be the same as a statement label, a block name, or a loop name outside these constructs.
Implementation Guideline: Check all nine combinations.
- T2. Check that a label, a block name, or a loop name in a nested subprogram, package, or task can be identical to a label, a block name, or a loop name outside such a construct. In particular, try a subprogram declaration in a block as well as a subprogram nested in a subprogram.
Implementation Guideline: Check all nine combinations.

T3. Check that a goto statement inside a loop, a block body, or an exception handler (of a block) is permitted to access a labeled statement appearing before and after the loop or block, respectively (see IG 5.9/T2).

T4. Check that a goto statement in a nested subprogram, a package, or a task body is not permitted to reference a statement label in the outer subprogram, package, or task body, nor can a goto statement in an exception handler transfer control into its associated body (see IG 5.9/T1).

Check that an exit statement in a nested subprogram, a package, or a task body is not permitted to reference a loop name in the outer subprogram, package, or task body (see IG 5.7/T1).

T5. Check that a loop parameter can be spelled the same as a label occurring prior to the loop but it cannot be the same as a label occurring inside the loop.

Check that a loop parameter can have the same name as the loop, but that an exit statement within the loop cannot then reference the loop name.

Check that if a loop name is hidden, it cannot be used in an exit statement within the loop.

T6. Check that a statement label, a block name, or a loop name in a loop, an accept statement, a block body, or an exception handler cannot be the same as a block identifier, a loop identifier, a variable, a constant, a formal parameter, a generic formal parameter, a named literal, a subprogram, an enumeration literal, a type, an entry, a package, an exception, or a generic unit declared in the enclosing body, and if the label is the same as a predefined exception or a user-defined exception defined in an enclosing unit, no handler for that exception can be written.

Implementation Guideline: Check for each form of declarative region. See also IG 8.3.f/T2.

Check that a label, a block name, or a loop name can be the same as the name of a derived enumeration literal or subprogram, and that the label is declared at the end of the declarative part.

T7. Check that a statement label in a subprogram body, a task body, or a package body cannot be the same as a block identifier, a loop identifier, a variable, a constant, a named literal, a subprogram, an enumeration literal, a type, an entry, package, an exception, or a generic unit declared in the subprogram (including formal parameters), the package (specification and body), or the task (specification and body).

T8. Check that a statement label declared outside a block can have the same identifier as an entity declared in the block, but a goto statement using that label is illegal within the block and legal outside the block.

T9. Check that in a nested body, N, an attempt to reference an entity, E, declared in an enclosing body is not legal if N contains a label, block name, or loop name E.

Implementation Guideline: The attempt to reference the outer declaration should occur before the attempt to declare a label, block name, or loop name.

8.3.b Loop Parameters

Semantic Ramifications

S1. There are several consequences of the rule that the scope of a loop parameter extends only to the end of the corresponding loop:

1. The value of the loop parameter can be accessed only within the loop body.

2. Nested loops can have identically named loop parameters, in which case the outer loop parameter is not directly visible within the inner loop. If the outer loop is labeled, selected component notation can be used.
3. Non-nested loops can use the same identifier for their loop parameters but these identifiers are distinct loop parameters and can have different types.
4. A loop parameter can have the same identifier as some entity declared in the immediately enclosing scope.

Test Objectives and Design Guidelines

- T1. Check that the value of a loop parameter cannot be accessed from outside the loop body.
Implementation Guideline: Attempt to access the loop parameter after the loop body and in a program containing no other use of the loop parameter identifier.
- T2. Check that:
- a. nested loops can have identically named parameters with or without distinct types, and references in the innermost loop are associated with the innermost parameter, etc.
 - b. non-nested loops can have identically named loop parameters with distinct types, and that references within each loop are to its own loop parameter;
 - c. a loop parameter can have the same identifier as a variable declared in the scope immediately containing the loop.

Implementation Guideline: Use loop parameters containing more than a single letter, since JOVIAL gives single letter loop parameters the Ada interpretation, but gives a different interpretation to multiple letter loop parameters.

8.3.c Records

Semantic Ramifications

- S1. Components of a given record type definition can have the same name as components of another record type definition, i.e., two record types can both have components named X.
- S2. The visibility rules for records imply that outside the record, the name of a component begins with the name of the object containing the component, and similarly, for components of components. Hence, if R is a record containing component S which contains component T, the only way to reference T is by writing R.S.T; R.T, T, and S.T are not legal references to this component.

Legality Rules

- L1. A record component identifier within a given record type definition must be uniquely named. In particular, components of different variants of a given record cannot have identical names (RM 3.7/3).

Test Objectives and Design Guidelines

- T1. Check that two components of a given record definition must be uniquely named. In particular, check that components of different variants cannot have the same name, even if they have the same, statically defined, subtype. Since record discriminants are considered components of a record (see RM 3.7.1/1), check also that duplications are not permitted among the names of discriminants and the names of any other record components of a given record definition.

Check that component names may be the same in separate record type definitions.

Check that component names may be the same as names of other objects, viz., formal parameters, labels, loop parameters, variables, constants, subprograms, packages, tasks, and types.

- T2. Check that partial names for record components (as in PL/I) are not permitted, e.g., that if a record component is named T, and this is the only declaration for identifier T, the name T is not permitted outside the record as a reference to this component. Also, for a record of records, if the full name for a component is R.S.T, check that R.T is not permitted as a valid reference, assuming (of course) that there is no T component of R.

8.3.d Enumeration Literals

Semantic Ramifications

S1. A homograph of an enumeration literal is not allowed in the same declarative region if the enumeration literal was declared with an enumeration literal specification (AI-00330). For example,

```
type ENUM is (RED, GREEN, BLUE);
function RED return ENUM;           -- illegal
```

S2. On the other hand, if an enumeration literal is implicitly declared by a derived type declaration, then a derived subprogram homograph can be (implicitly) declared in the same declarative region. An explicit declaration of a function homograph is also allowed:

```
package P is
  type ENUM is (RED, GREEN);
  type T is private;
  function RED return T;
private
  type T is new ENUM;
  -- derives enumeration literal RED
  type NT is new T;
  -- derives enumeration literal RED and function RED
end P;
```

The explicit declaration of function RED hides the enumeration literal derived by T's full declaration, and so T's full declaration is legal. Similarly, the declaration of NT is legal because the derived function RED hides the implicitly declared enumeration literal (see AI-00002 and RM 8.3/17).

Legality Rules

- L1. Duplicate enumeration literals (including character literals) are not permitted in a given enumeration type definition (RM 8.3/15 and RM 8.3/17).
- L2. A function homograph of an enumeration literal declared by an enumeration type definition is not allowed in the same declarative region as the enumeration type definition (RM 8.3/17 and AI-00330).

Test Objectives and Design Guidelines

- T1. Check that duplicate enumeration literals (including character literals) are not permitted in a single enumeration type definition (see IG 3.5.1/T3).

Implementation Guideline: Use duplicates which differ only in the case of the letters as well as lexically identical literals.

- T3 Check that an explicitly declared function homograph for an enumeration literal cannot be given in the same declarative region as the enumeration literal if the enumeration literal is declared by an enumeration literal specification (see IG 8.3/T1).

Check that a derived subprogram can hide a derived enumeration literal (see IG 8.3/T32).

8.3.e Subprogram and Entry Parameters

Semantic Ramifications

- S1. The use of a formal parameter in a named association does not hide an entity with the same name used as the actual parameter, e.g., calls of the form $F(A \Rightarrow A)$ are permitted, where the first A is F 's formal parameter name and the second A names an identifier visible in the context of the call to F . The first occurrence of A names the formal parameter by selection (RM 8.3/11); the second occurrence names a directly visible A .

Legality Rules

- L1. Formal parameters of subprograms must be distinct from each other and from identifiers declared in the subprogram's declarative part (RM 8.1/2, RM 8.3/15, and RM 8.3/17).
- L2. A parameter name cannot be used in a default value appearing later in a parameter list (RM 6.1/5).

Test Objectives and Design Guidelines

- T1. Check that a subprogram specification cannot have duplicate formal parameter names.

Implementation Guideline: Check for function and procedure specifications in a generic and nongeneric subprogram declaration, a subprogram body (no preceding declaration), a generic formal subprogram declaration, a renaming declaration, and a body stub. Some forms of this test should use more than two formal parameters, with the duplicate parameter names occurring at different positions and in different identifier lists.

Check that a formal parameter of a subprogram or generic subprogram cannot have the same identifier as one declared by a local declaration in the subprogram body or as one declared as a generic formal parameter.

Implementation Guideline: The local declaration should attempt to declare a: variable, constant, named constant, exception, type, subtype, package, task unit, generic unit (package, function, and procedure), block name, loop name, statement label, function, and procedure.

Implementation Guideline: Repeat the test for generic and nongeneric subprogram bodies.

- T2. Check that within a subprogram or an accept statement, a formal parameter can be used directly in a range constraint, a discriminant constraint, an index constraint, and an exception handler, but it cannot be used in a default value appearing in the parameter list (see IG 6.1/T11).
- T3. Check that a formal parameter in named parameter association is not confused with an actual parameter identifier having the same spelling.
- Implementation Guideline:* Include a check for subprogram and entry calls. Include subprograms declared by instantiation.
- T11. Check that an entry declaration cannot declare an entry or an entry family with duplicate formal parameter names.

8.3.1 Packages

Semantic Ramifications

S1. Because the declarations in a package specification belong to the same declarative region as the declarations given in the declarative part of the body, the package specification's declarations are always visible in the package body. For example:

```
package OUTER is
  package P is
    X : INTEGER;
  end P;
end OUTER;

package body OUTER is
  X : FLOAT;
  package body P is
    -- reference to X is to INTEGER X, not FLOAT X.
  begin
    null;
  end P;
end OUTER;
```

Test Objectives and Design Guidelines

- T1. Check that inside a nested package body, an attempt to reference an identifier declared in the corresponding package specification is successful even if the same identifier is declared in the outer package body or the outer package specification. Check that the outer declaration is visible by selection after to the inner declaration and is directly visible prior to the inner declaration.

Implementation Guideline: See the Guidelines for IG 8.3/T21

Implementation Guideline: Include a check for generic packages as well as for nongeneric.

- T2. Check that the visible and private parts of a package specification form a single declarative region that is shared with the declarative part of the package body. In particular, check that two homographs cannot be declared within the following parts of the declarative region: visible-visible, visible-private, visible-body, private-private, private-body, and body-body.

Implementation Guideline: Check for generic packages and subunits as well.

Check that a label defined in a package body cannot be identical to an identifier declared in the corresponding package specification or body (see IG 8.3.a/T1).

Check that if a package and an entity declared in the package specification have the same name, the entity can be used in the package body without being confused with the package name.

Implementation Guideline: In particular, check the declaration of a private type.

- T3. Check that, if a package body is nested inside a package body, the inner package body can contain a label identical to a label in the outer package body or to an identifier declared in the outer package body or its specification.

8.4 Use Clauses

Semantic Ramifications

S1. If subprograms with the same parameter and result type profile are declared in different packages, both subprograms are made visible by a use clause:

```

package P is
  procedure P1 (X : INTEGER);
end P;

package Q is
  procedure P1 (Y : INTEGER);
end Q;
use P, Q;
...
P1 (3);           -- ambiguous
P1 (Y => 3);       -- unambiguous

```

S2. The immediate scope of a library package identifier mentioned in a with clause extends throughout the compilation unit associated with the with clause (RM 8.6/2, RM 8.2/2, and RM 10.1.1/5). Hence, any potentially visible declaration having the same identifier as the package identifier is not made visible by a use clause (RM 8.4/5):

```

package P is
  procedure P;
end P;

package Q is
  procedure P (X : INTEGER);
end Q;

with P, Q;
procedure R is
  use Q;      -- Q.P is not made directly visible
  use P;      -- P.P is not made directly visible

```

S3. No potentially visible names are made directly visible until after the end of the use clause. Hence, use P; use Q; can be legal when use P, Q; would be illegal:

```

package P is
  package Q is
    ...
  end Q;
end P;
use P, Q;      -- illegal (1)
use P; use Q;  -- legal (2)

```

(1) is illegal since Q is not directly visible prior to the use clause. (2) is legal because after use P is elaborated, Q becomes visible, making use Q legal.

S4. Although a use clause is usually thought of as making certain declarations directly visible that were not previously visible, a use clause can also cause previously visible declarations to become invisible. For example:

```

package P1 is
  X : INTEGER;
end P1;

package P2 is
  procedure X;
end P2;

use P1;           -- makes INTEGER X visible
use P2;           -- no X is directly visible now

```

After the second use clause, two declarations of X are potentially visible. Since one of these is not the declaration of a subprogram or an enumeration literal, RM 8.4/6 says *neither* declaration is made visible.

S5. The effect of withdrawing visibility can also occur for a package, so use P, P is not always equivalent to use P; use P:

```

package OUTER is
  package P is
    P : INTEGER;
  end P;
end OUTER;

package ONE is
  use OUTER;       -- OUTER.P directly visible now
end ONE;

package TWO is
  use OUTER.P;     -- OUTER.P.P directly visible now
end TWO;

package THREE is
  use OUTER;
  use P;           -- means use OUTER.P
                  -- no P is directly visible now
end THREE;

```

The situation for package ONE is conventional. For package TWO, since the immediate scope of OUTER.P does not include the use clause, OUTER.P.P is made directly visible, but OUTER.P is not directly visible. In package THREE, the first use clause makes OUTER.P directly visible. The elaboration of the second use clause makes two P's potentially visible. Since neither of the potentially visible P's are subprograms, neither P is actually made directly visible. Note that if use P were replaced with use P, P, there would be no change in effect, but for the sequence, use P; use P, the second use P would be illegal since no P is directly visible after the first use P.

S6. The effect of a use clause in the visible part of a package specification extends over the package body, but not otherwise outside the package:

```

package P is
  use LIB_Q;
  X : INTEGER;
end P;

-- (1)

```

```

package body P is
    ... -- (2)
end P;

```

Although the scope of P.X includes (1), the scope of the use clause does not. Hence, the names made visible by use LIB_Q are not also made visible at (1), but are visible at (2).

S7. Consider the following declarations:

```

procedure Q1 (X : STRING);
package R1 is
    procedure Q1 (X : STRING);
end R1;

package R2 is
    procedure Q1 (X : FLOAT);
end R2;
use R1;           -- (1)
use R2;           -- (2)
...
procedure Q1 (Y : FLOAT);  -- R2.Q1 no longer visible

```

The use clause at (1) does not make R1.Q1 visible since (1) is within the immediate scope of the first declaration of Q1, and R1.Q1 is a homograph of this declaration (RM 8.4/5). Moreover, RM 8.4/6 does not apply since there is only one potentially visible Q1. The use clause at (2), however, makes two declarations of Q1 potentially visible, and since both are declarations of subprograms, RM 8.4/6 says that both declarations are made visible. But RM 8.4/5 still forbids making R1.Q1 directly visible; hence, Q1("AB") will be an unambiguous call. R2.Q1 will be made visible, however, until the later declaration of a homograph for R2.Q1.

S8. A renaming declaration is not treated any differently from any other declaration with respect to a use clause, even if the same entity is renamed with the same identifier:

```

package P1 is
    USE_ERROR : exception renames TEXT_IO.USE_ERROR;
end P1;

package P2 is
    USE_ERROR : exception renames TEXT_IO.USE_ERROR;
end P2;

use P1, P2;

```

The use clause does not make USE_ERROR visible, even though P1.USE_ERROR and P2.USE_ERROR denote the same exception.

S9. RM 8.4/3 requires consideration of the set of packages named in use clauses. If a package is renamed, use of the new name has the same effect as use of the original name:

```

package NP renames P;
use NP, P;           -- equivalent to use P, P;

```

Since the set of named packages is considered, and since sets do not contain duplicates, use NP, P is equivalent to use P, P, which is equivalent to use P.

S10. Since entries are not declared immediately in the visible part of packages, they are not

among the potentially visible entities that can be made visible by a use clause. Of course, if an entry is renamed as a procedure in the visible part of a package, the new name becomes a potentially visible entity.

S11. Use clauses combined with derived types can have some possibly surprising effects:

```
package MATRIX_OPS is
  type MATRIX is array ...;
  function "+" (L, R : MATRIX) return MATRIX;
end MATRIX_OPS;

procedure P is
  type MY_MATRIX is new MATRIX_OPS.MATRIX;
  package Q is
    procedure INVERSE (X : in out MY_MATRIX);
  end Q;
  X : MY_MATRIX;
  use Q;
  package body Q is ... end Q;
begin
  INVERSE (X);  -- calls Q.INVERSE
end;
```

Now suppose we add a declaration of INVERSE to MATRIX_OPS and recompile. The call to INVERSE in P will now call the derivation of P.INVERSE that is implicitly declared as an operation of MY_MATRIX. Since this implicit declaration declares a homograph of Q.INVERSE, the use clause no longer makes Q.INVERSE directly visible.

S12. If a use clause names an enclosing package, it is important that none of the declarations in the named package be considered potentially visible:

```
package P is
  X : FLOAT;
end P;

package Q is
  X : INTEGER;
  use P;      -- P.X potentially visible
  use Q;      -- Q.X still visible
```

Since the use Q is, in effect, ignored, Q.X is not in the set of potentially visible declarations together with P.X, and hence Q.X remains directly visible. (If Q.X were considered potentially visible, then RM 8.4/6 would imply that neither P.X nor Q.X would be directly visible.)

Changes from July 1982

S13. Potentially visible subprograms declared by a renaming declaration, a generic instantiation, or an implicit declaration are now covered by the rules for making such subprograms visible.

Changes from July 1980

S14. The scope of a use clause is defined.

S15. The rules now consider the set of packages named in use clauses rather than the set of package names, so renamed packages are handled correctly.

S16. Enumeration literals are made directly visible just like subprograms.

S17. Two subprogram homographs can now be made visible by a use clause.

S18. Consideration of the effect of a use clause is no longer delayed when resolving overloaded names.

Legality Rules

L1. The names given in a use clause must denote packages.

L2. If a use clause appears in a context clause, the names must be simple names denoting library packages named in previous with clauses of the context clause (RM 10.1.1/3).

Test Objectives and Design Guidelines

T1. Check that the name in a use clause must be the name of a library package named in a previous with clause of the context clause.

Implementation Guideline: In particular, check that it cannot be the name of a library subprogram, the name of a generic package, or the name of a package made visible by a use clause. Nor can it be the name of a package in a with clause that applies to the unit.

Implementation Guideline: Check that use clauses can name packages declared by package declarations, generic instantiations, and renamings of packages.

Check that when a use clause appears in a context clause, it must name library packages only, and the names must all be simple names (see IG 10.1.1/T1).

T2. Check that if a use clause names an enclosing package, the use clause has no effect.

Implementation Guideline: In particular, check that previously visible declarations remain visible.

Implementation Guideline: Check the effect within a subunit of the named package.

Check that if a declaration is directly visible prior to the occurrence of a use clause, and is not in the set of potentially visible declarations, it remains directly visible after the use clause.

Implementation Guideline: In particular, check that if a subprogram, Q, is already directly visible and a group of potentially visible subprograms includes a homograph for Q, the homograph for Q is not made visible. Include a case where the homographs have different formal parameter names.

Check that if a homograph for a potentially visible subprogram or object is declared after a use clause, the potentially visible entity is no longer visible.

Check that the effect of use P can be to make P invisible.

T4. Check that if the set of potentially visible declarations includes a mixture of subprogram or enumeration literal declarations together with a declaration of some other kind of entity, none of the declarations are made visible.

Implementation Guideline: Include cases in which a single use clause makes all the declarations visible, and a case in which a sequence of use clauses makes some of the declarations visible first, and later use clauses retract the visibility.

T5. Check that two potentially visible homographs of a subprogram identifier can be made directly visible by a use clause.

Implementation Guideline: Visibility can be checked by providing the subprograms with differently named formal parameters.

T6. Check that if two renaming declarations (in different packages) declare the same identifier and both declarations rename the same entity, a use clause cannot make the identifier visible.

Implementation Guideline: In particular, check exception names declared by TEXT_IO and an instantiation of SEQUENTIAL_IO or DIRECT_IO.

T7. Check that the names made visible by a use clause are not made visible until after the end of the use clause.

- T8. Check that the scope of a use clause in the visible part of a package does not extend outside the package except for the package body.
- T9. Check that a use clause makes implicitly and explicitly declared operators visible except when there is a homograph of an operator already directly visible.

8.5 Renaming Declarations

Semantic Ramifications

- S1. In an object renaming declaration, the evaluation of the object name determines which object is denoted by the new name. Subsequent assignments to the renamed object will change the value associated with the new name:

```

type ACC_STR is access STRING;
XAS : ACC_STR := new STRING' ("ABCD");
I   : INTEGER := 2;
X1  : STRING renames XAS (I..4);           -- X1 = "BCD"
-- note use of an unconstrained array type STRING as the type mark
Y1  : CHARACTER renames XAS(2);           -- Y1 = 'B'
...
XAS.all := "EFGH";

```

After the assignment, X1 = "FGH" and Y1 = 'F'. Similarly,

```

X1 := "JKL";

```

implies that XAS.all = "EJKL" and Y1 = 'J'. Since the object name is evaluated when the renaming declaration is elaborated, subsequent changes to I do not affect the value associated with X1. Even after I is changed to 1, for example, X1'LENGTH will still equal 3, and X1'FIRST will equal 2. Similarly, the object denoted by X1 depends on the value of XAS at the time X1's declaration is elaborated; subsequent changes to XAS do not affect the object denoted by X1 and Y1:

```

-- X1 = "JKL"
XAS := new STRING' ("WXYZ");
-- X1 is still "JKL"

```

- S2. A function call yields a value (RM 4.4/3), not an object, and a subcomponent of a function value is also a value. Hence, a subcomponent of a function result cannot be renamed. However, a function call can appear in the prefix of a renamed entity's name if a portion of the prefix containing the call denotes an access value, so the name itself denotes an object designated by an access value, or a subcomponent of such an object:

```

function F returns ACC_STR;
...
-- assume F.all = "ABCD"
X2 : STRING renames F.all;           -- X2 = "ABCD"
Y2 : CHARACTER renames F(2);         -- Y2 = 'B'

```

- S3. Any constraint in the type mark of a renaming declaration is ignored; the new entity has the subtype of the renamed object:

```

subtype S1_3 is STRING(1..3);
X3 : S1_3 renames XAS(2..4);

```

X3'FIRST is 2, not 1, since XAS(2..4)'FIRST = 2.

S4. It is possible for more than one object to have the required type if the prefix of an object name contains an overloaded function call:

```
function F return ACC_STR;
function F (X : INTEGER := 1) return ACC_STR;
...
X4 : STRING renames F(2..4); -- illegal
```

The function call, F, is ambiguous since F can denote either the parameterless F or the F with a single default parameter. Either function returns a suitable access value. Now consider the following functions:

```
function G return ACC_STR; -- G1
function G return STRING; -- G2
...
X5 : STRING renames G(2..4); -- illegal
```

The object name is illegal since either G1 or G2 can be sliced and both slices have type STRING. It is irrelevant that G2(2..4) does not denote an object, but instead denotes a value, and so is illegal. Such information cannot be used to resolve the function call (RM 8.7/7-13).

S5. The properties of a new object name are, generally speaking, those of the renamed object. In particular, a formal parameter of a subprogram, entry, or generic unit can be renamed within the unit. If the parameter is of mode in, then the new name is treated as a constant. Similarly, a renaming of an in out parameter is a variable. A renaming of an out parameter is also a variable, but the new name has the properties of an out parameter, i.e., it cannot be passed as an in out actual parameter, cannot be read, etc. (see IG 6.2/S). If a renamed constant is static, however, the new name is not static, since RM 4.9/6 explicitly limits static constants to those declared by a constant declaration.

S6. If the restrictions on the renaming of a subcomponent dependent on a discriminant did not exist, it would be possible to assign a value to the containing variable that eliminates the renamed object:

```
type REC (D : INTEGER := 1) is
  record
    S : STRING (1..D);
    case D is
      when 4 =>
        A : INTEGER range 0..10;
        AS : STRING(1..3);
      when others =>
        B : FLOAT;
    end case;
  end record;
X : REC := (4, "ABCD", 5, "DEF");
XR : INTEGER renames X.A; -- illegal
XS : STRING renames X.S; -- illegal
XSC : CHARACTER renames X.S(4); -- illegal
XASC : CHARACTER renames X.AS(3); -- illegal
...
X := (3, "XYZ", 1.5);
```

After the assignment to X, X.A, X.S(4), and X.AS(3) no longer exist. No problems arise since XR, XSC, and XASC are not declared legally. XS is also illegal since an implementation may choose to store components like X.S in physical locations that change when the discriminant

value changes. These restrictions allow an implementation to represent renamings of objects in terms of an address pointing to the renamed object, where the "address" specifies the starting bit of the renamed object. (Note that specifying an address less precise than the starting bit will not, in general, be adequate for renaming packed array or record components, since, for example, such components might not be aligned on byte boundaries. Note also that the offset within an addressable unit cannot always be computed at compile time, e.g., for a packed BOOLEAN array component A(I).)

S7. A variable having an unconstrained type with default discriminants can be declared directly in an object declaration or indirectly as a component of a record or array variable:

```

type REC2 is
  record
    V : REC;
  end record;
type ARR is array (1..2) of REC;
Y : REC2;
Z : ARR;

```

The variables Y, V and Z(1) can be renamed, but no subcomponent of these variables can be renamed, e.g., Z.V.A and Z(1).A cannot be renamed.

S8. Any subcomponent can be renamed if the object containing the subcomponent is a constant or is an object designated by an access value. In addition, any subcomponent of a nongeneral variable can be renamed if discriminants of the containing variable do not have defaults, since such objects cannot have their discriminant values changed (i.e., in terms of the RM, such objects cannot be declared with an unconstrained type in an object declaration, a component declaration (of a record), or a component subtype indication (of an array)).

S9. A renamed package acts like the original package name except that the new name cannot be used in the prefix of an expanded name (see IG 4.1.3/S for further discussion of this restriction).

S10. One of the criteria determining whether an expression is static is that all operators or operator symbols in the expression must denote predefined operators (RM 4.3/2, RM 4.9/2). Consequently, a renamed operator cannot always be used in a static expression.

```

function PLUS (L, R : INTEGER) return INTEGER is
function "+" (L, R : INTEGER) return INTEGER is
  PLUS (3, 4) ...           -- not static (1)
  3 + 4 ...                 -- still static (2)
  "+" (3, 4) ...            -- still static (3)

```

(1) is not static since PLUS is not an operator symbol. (2) and (3) are static since "+" is an operator and the new "+" denotes STANDARD. "+". (The evaluation of PLUS in the second renaming declaration determines that PLUS denotes the entity STANDARD. "+", so does the new "+".)

S11. RM 8.5/7 says, in effect, that the normal overloading resolution process applies to the subprogram or entry name in a subprogram renaming declaration, except that the names of the formal parameters are not considered. If the parameter and result type of the procedure suffice to resolve the subprogram name, the renaming declaration is illegal. (Note that neither the formal parameter names, the presence or absence of default expressions for the parameter modes, are used to resolve the name (RM 6.6/6). After the name has been resolved, however, the parameter modes must be the same at corresponding parameter positions.)

```

procedure P (X : INTEGER);
procedure P (X : FLOAT);

```

```

procedure NP1 (Y : in INTEGER) renames P;      -- legal
procedure NP2 (Z : in out INTEGER) renames P;  -- illegal; mode

```

To show that modes are not used to resolve the names, two subprograms or entries with identical parameter and result type profiles must be visible:

```

package P1 is
  procedure Q (X : INTEGER);
end P1;

package P2 is
  procedure Q (X : in out INTEGER);
end P2;

use P1, P2;

procedure NQ (X : INTEGER) renames Q;          -- illegal

```

Although only one of these Q's can legally serve as the renamed entity, this fact cannot be used to decide which subprogram is to be renamed.

Similar examples can be constructed for entries:

```

task type T1 is
  entry Q (X : INTEGER);
end T1;

task type T2 is
  entry Q (X : in out INTEGER);
end T2;

function F return T1;
function F return T2;
...
procedure NQ (X : INTEGER) renames F.Q;        -- illegal

```

In this case, F cannot be resolved on the basis of Q's required parameter and result type profile.

S12. If a generic formal *In out* parameter is renamed, the constraints associated with the new name are those of the corresponding actual parameter, since any constraint associated with a formal *In out* generic parameter is ignored in the instantiated unit (RM 12.1/4).

Changes from July 1982

S13. A subprogram or package can now be renamed within the subprogram or the package itself.

Changes from July 1980

S14. Single tasks (i.e., tasks that do not have a user-defined task type) can no longer be renamed.

S15. Any constraints associated with the type mark in an object renaming declaration are ignored. In particular, the renamed object need not have the constraints of the type mark.

S16. The restriction on renaming components dependent on a discriminant is relaxed; such components can be renamed if the discriminants do not have defaults or if the renamed object is a constant, or is constrained. The restriction is extended to include formal generic in out parameters (whether constrained or not). In addition, the restriction now includes components that mention an enclosing discriminant in an index or a discriminant constraint.

S17. Formal parameter constraints and the return subtype, if any, in the specification of a renaming declaration need not be the same as those of the renamed subprogram or entry.

S18. The mode of a formal parameter is not used to decide which subprogram or entry is being renamed.

S19. A renamed entry cannot be used in contexts where only an entry call or an entry name is allowed.

Legality Rules

- L1. An object renaming declaration must rename an object. (In particular, the name in an object renaming declaration must not denote a value returned by a function call or a subcomponent of such a value.)
- L2. In an object renaming declaration, the renamed object must have the base type of the type mark.
- L3. If a variable, V1, has discriminants:
 - and is a generic formal parameter of mode in out, or
 - the discriminants have defaults, V1 has an unconstrained subtype, and V1:
 - is declared as a formal in out or out parameter of a subprogram or entry,
 - is declared by an object declaration;
 - is a component of an array variable, V2;
 - is a component of a record variable, V2;
 and the renamed object is a subcomponent of V1, the subcomponent must not be:
 - a component of V1's variant part (if any);
 - a component declared with an index constraint or a discriminant constraint if the constraint uses a discriminant of V1;
 - a subcomponent of one of the above components.
- L4. An exception renaming declaration must rename an exception.
- L5. A package renaming declaration must rename a package.
- L6. A function declared by a renaming declaration must rename a function having the same parameter and result type profile.
- L7. A procedure declared by a renaming declaration must rename a procedure or an entry having the same parameter and result type profile.
- L8. Corresponding parameters in a subprogram renaming declaration must have the same mode.
- L9. A renaming declaration with the designator "=" is only allowed to rename another equality operator (RM 6.7.5).

Test Objectives and Design Guidelines

- T1. Check that only an object can be renamed in object renaming declarations.

Implementation Guideline: Check that a subcomponent of a function value cannot be renamed as an object.

Check that the base type of the type mark and the base type of the renamed object must be the same.

Check that a subtype indication with a range constraint, accuracy constraint, index constraint, or discriminant constraint cannot be used in place of the type mark in an object renaming declaration.

- T2. Check that a renaming declaration is illegal if the renamed object is a:

- component of a variant part;
- a component declared with a discriminant constraint or an index constraint using a discriminant of an enclosing record type; or
- a subcomponent of the above kinds of components,

and the containing object has an unconstrained subtype with default discriminant values and is declared:

- in a nonconstant object declaration, or
- as a formal In out or out parameter of a subprogram or entry, or
- as a component of a record or array variable.

(Note: generic formal parameters are checked in T3).

Implementation Guideline: Include cases where the variable is a renaming of a suitable variable.

- T3. Check that a renaming declaration is illegal if the renamed object is

- a component of a variant part;
- a component declared with a discriminant or index constraint using a discriminant of an enclosing record type; or
- a subcomponent of the above kinds of components,

and the containing object is a formal generic parameter of mode In out having a:

- formal generic type (declared in the same or an enclosing generic unit);
- unconstrained nonformal type;
- constrained nonformal type.

Implementation Guideline: Try cases where the discriminants do and do not have default values.

Implementation Guideline: Include a case where the containing object is a renaming of a formal generic parameter.

- T4. Check that a renamed constant is considered a constant. Check constants declared as:

- a constant object,
- an In parameter of a subprogram or an entry,
- an In parameter of a generic unit,
- a discriminant of a record,

Implementation Guideline: Include a discriminant of a record variable declared with an unconstrained type, including a variable designated by an access value.

- a loop parameter,
- a deferred constant (after its full declaration),
- a renamed constant.

Implementation Guideline: Include checks that a subcomponent of a constant is a constant.

Implementation Guideline: Confirm constancy by attempting to assign to the new name or by passing the new name as an in out or out parameter of a subprogram or as an in out parameter of a generic unit. Note that the value of a renamed loop parameter or record discriminant can change even though such objects cannot be assigned to directly.

Check that renamings of the above forms of constant have the correct value.

Implementation Guideline: In particular, check that when a component of a generic array in parameter is renamed, the correct value is obtained for instantiations with different component types.

Implementation Guideline: Include checks that components dependent on a discriminant can be renamed (see T3 for further detail).

T5. Check that a variable created:

- by an object declaration,
Implementation Guideline: Include a variable declared in a package.
- as a subprogram or entry in out formal parameter (out parameters are checked in T7),
- as a generic in out formal parameter, and
- by an allocator,

can be renamed and has the correct value. Check that the new name can be used in an assignment statement and passed as an actual subprogram or entry in out or out parameter, and as an actual generic in out parameter.

Implementation Guideline: Use scalar, composite, access, private, and task objects.

Implementation Guideline: Include cases where the type mark is an unconstrained array type or an unconstrained type whose discriminants do not have defaults.

Check that if the value of the renamed variable is changed, the new value is reflected by the value of the new name.

For a renamed variable designated by an access value, check that a change in the access value does not affect which variable is denoted by the new name.

Implementation Guideline: Check that the prefix of the renamed variable can be a function call.

Check that any subtype constraint imposed by the type mark used in the renaming declaration is ignored, and the subtype constraint associated with the renamed variable is used instead.

Implementation Guideline: The type mark should impose constraints both wider and narrower than those of the renamed variable.

Implementation Guideline: For renamings of a generic formal in out parameter, check that the new name has the constraint of the actual parameter, not the formal parameter.

T6. Check that a component or a slice of a variable created:

- by an object declaration,
- as a subprogram or entry in out formal parameter (out parameters are checked in T7),
- as a generic in out formal parameter,
- by an allocator,

can be renamed and has the correct value. Check that the new name can be used in an assignment statement and passed as an actual subprogram or entry in out or out parameter, and as an actual generic in out parameter.

Implementation Guideline: Check components having a scalar, composite, access, private, and task type.

Check that if the value of the renamed variable or component is changed, the new value is reflected in the value of the new name.

Check that a renamed slice can be sliced and indexed for purposes of assignment and to read the value.

Check that any constraint imposed by the type mark used in the renaming declaration is ignored, and that the subtype constraint associated with the renamed variable is used instead.

Implementation Guideline: The type mark should impose constraints both wider and narrower than those of the renamed variable.

Implementation Guideline: For renamings of a generic formal in out parameter, check that the new name has the constraint of the actual parameter, not the formal parameter.

- T7. Check that any renaming of an out formal parameter of a subprogram or entry satisfies the usual rules for out parameters, namely:

Implementation Guideline: Include renamings of components of an out parameter.

Implementation Guideline: Include renamings of renamings in some cases.

Check that a renamed out parameter, an out parameter component, or an out parameter slice can be assigned to.

- T8. Check that an exception renaming declaration can only rename an exception.

- T9. Check that exceptions can be renamed.

Check that renamed exceptions cannot be used together with the original exception in the same exception handler (see IG 11.2/T2).

- T10. Check that only packages can be renamed in a package renaming declaration.

Implementation Guideline: Check that generic packages cannot be renamed.

- T11. Check that packages can be renamed, and that the new name for a package can be used in a renaming declaration.

Implementation Guideline: Check that a package can be renamed inside itself. Include a renaming of a generic package inside itself.

- T12. Check that an entry or procedure cannot be renamed as a function.

Check that an entry family cannot be renamed.

Check that a function cannot be renamed as a procedure.

Check that a parameterless function cannot be renamed as an object.

Check that a renaming is illegal if the parameter modes are not the same, and check that the modes are not used to help resolve which subprogram or entry is being renamed.

Check that a subprogram or an entry must have the correct number of formal parameters.

Implementation Guideline: In particular, check that the renaming is illegal even if omitted or extra parameters appear at the end and have default values.

Check that corresponding parameters (or the result type) cannot have different base types (even if the types are convertible).

- T13. Check that a subprogram or an entry can be renamed with:

- different parameter names;
- different default values;
- different parameters having default values;

and that the new names/defaults are used when the new name is used in a call.

Implementation Guideline: Include a case where the formal parameter names are just reordered, to ensure that the new formal names are not associated with the old parameter positions.

Check that formal parameter constraints (or result type constraints) for the new name are ignored in favor of the constraints associated with the renamed entity.

Check that a subprogram or an entry can be renamed within its own body.

Implementation Guideline: Include a renaming within a generic unit.

Check that the new name can be used in a renaming declaration.

Check that when default expressions are aggregates, the legality of the aggregate depends on the type mark used in the renamed program, not the type mark used in the renaming declaration. For example, check that use of `others` in a named association requires a static index subtype in the renamed subprogram's formal parameter.

T14. Check that the number of formal parameters is used to determine which subprogram or entry is being renamed.

Check that the base type of the formal parameter and the result type (if present) is used to determine which subprogram or entry is being renamed.

Check that the presence or the absence of a result type is used to identify which subprogram or entry is being renamed.

T15. Check that a renamed entry cannot be used in a conditional entry call, a timed entry call, or as the prefix to the `'COUNT` attribute.

T16. Check that the attributes `'SUCC`, `'PRED`, `'IMAGE`, and `'VALUE` can be renamed.

T17. Check that a renaming of a predefined operation cannot be used in a static expression if the new name is an identifier (see IG 4.9/T8).

Check that a further renaming as an operator symbol allows the new name to be used in a static expression.

8.6 The Package Standard

Semantic Ramifications

S1. RM 8.6/2 says that the declaration of a library unit is assumed to occur immediately within `STANDARD`. Suppose we compile library packages `LIB_UNIT` and `NEXT_UNIT`, in that order. The wording of RM 8.6/2 might seem to suggest that after compilation, both units would be included in `STANDARD`, e.g.:

```
package body STANDARD is
  package LIB_UNIT is
    P : INTEGER;
  end LIB_UNIT;
```

```

package NEXT_UNIT is
    ...
end NEXT_UNIT;

package body NEXT_UNIT is
begin
    STANDARD.LIB_UNIT.P := 3;      -- not legal
end;
end STANDARD;

```

If STANDARD were truly a normal package, the reference to variable P would be legal. But RM 8.6/2 imposes special visibility rules so units like LIB_UNIT are not visible from within NEXT_UNIT. In particular, LIB_UNIT is not visible because no with clause naming LIB_UNIT applies to NEXT_UNIT. Since no with clause applies to NEXT_UNIT's body, it is best to think of STANDARD as not even containing LIB_UNIT when NEXT_UNIT's body is compiled.

S2. Now consider the difference between dependence on a library unit and the applicability of a with clause to a library unit:

```

package P is                                -- library unit
    OBJ1 : INTEGER := 5;
end P;

with P;
package Q is                                -- library unit
    OBJ2 : INTEGER := P.OBJ1;
end Q;

package body Q is                            -- with P is applicable
    OBJ3 : INTEGER := STANDARD.P.OBJ1;
end Q;

with Q;
package R is                                -- only depends on P
    OBJ4 : INTEGER := STANDARD.P.OBJ1;      -- illegal
end R;

```

The with clause given for package Q applies to P's body, so the use of the name STANDARD.P is allowed there. Package R depends on package P (in the sense that recompilation of package P will make package R obsolete), but there is no with clause for P that applies to R. Hence the name STANDARD.P is not allowed in package R.

S3. The predefined package STANDARD is not a library unit (RM C/22), so a library unit having the name STANDARD can be compiled. If such a unit is a package, it does not replace the declaration of STANDARD; it is merely a library unit named STANDARD that is declared within the predefined STANDARD package. Of course, the usual visibility rules apply within such a user-defined library unit, i.e., within the unit, the name STANDARD refers to the unit, not the predefined package. Similarly, with clauses that mention STANDARD refer to the library unit, not to the predefined package. For example:

```

package STANDARD is                          -- library unit
    OBJ1 : INTEGER := 5;
    OBJ2 : INTEGER := STANDARD.OBJ1;         -- user-defined STANDARD
end STANDARD;

```

```

with STANDARD;                                -- user-defined STANDARD
package Q is                                   -- library unit
    OBJ2 : INTEGER := STANDARD.OBJ1;
    OBJ2A : STANDARD.BOOLEAN;                -- illegal
    OBJ2B : STANDARD.STANDARD.BOOLEAN;       -- illegal
end Q;

package body Q is
    OBJ3 : INTEGER := STANDARD.OBJ1;
end Q;

with Q;
package R is
    OBJ4 : INTEGER := STANDARD.OBJ1;         -- illegal
    OBJ5 : STANDARD.BOOLEAN;                 -- predefined package
end R;

```

In the declaration of Q.OBJ2A, STANDARD.BOOLEAN is illegal because no declaration of the identifier, BOOLEAN, occurs in the user-defined package STANDARD. Similarly, the name STANDARD.STANDARD.BOOLEAN is illegal because STANDARD is not an identifier declared in user-defined package STANDARD. In the declaration of OBJ4 and OBJ5, STANDARD denotes the predefined package since the user-defined library unit is not visible.

S4. STANDARD is not a reserved word. Hence, within a declarative part, the following declarations are permitted:

```

package P is                                   -- library unit
    type T is
        record
            INTEGER : FLOAT;
        end record;
    STANDARD : T;
end P;

```

Within P's declarative region and after the declaration of the object, STANDARD, the name STANDARD.INTEGER refers to the component of the record object. STANDARD.P.STANDARD.INTEGER is an illegal name, since 1) the first identifier in a selected component name refers to a directly visible identifier, 2) only the record object STANDARD is directly visible, and 3) there is no P component in the record object.

S5. Since the name of a library unit is implicitly declared in STANDARD, the name cannot be a homograph (see RM 8.3/15) of a name that is already declared in package STANDARD. In particular, a library unit cannot have the name BOOLEAN, INTEGER, FLOAT, CHARACTER, ASCII, NATURAL, POSITIVE, STRING, DURATION, CONSTRAINT_ERROR, NUMERIC_ERROR, PROGRAM_ERROR, STORAGE_ERROR, or TASKING_ERROR. Similarly, no library unit package can have the name TRUE or FALSE, but a library unit subprogram can have the name TRUE or FALSE as long as it is not a homograph of the enumeration literals TRUE or FALSE (i.e., as long as it is not a parameterless function with return type STANDARD.BOOLEAN).

S6. Since the names of nongraphic characters are given in *italics* in Appendix C of the RM, these names are not considered to be declared in STANDARD, so a library unit having the name NUL, for example, can be declared.

S7. Operators cannot be declared as library units because the designator of a library unit must be an identifier (see RM 10.1/3).

S8. When compiling a subunit, the RM states (RM 10.2/6) that "visibility within the proper body of a subunit is the visibility that would be obtained at the place of the corresponding body stub." Consequently, the ancestor *library* unit is implicitly declared in STANDARD. Hence, within the subunit, unique selected component names starting with STANDARD can be used to designate identifiers in any ancestor unit (if there are no visible user declarations of STANDARD). For example:

```

with Q;
procedure P is
  I : INTEGER;
  procedure Q is separate;
begin
  ...
end;

separate (P)
procedure Q is ... end Q;

```

Within procedure body Q, the name STANDARD.P refers to the parent procedure, the name STANDARD.P.I refers to the variable declared in P, and the name STANDARD.P.Q.x could be used to refer to an identifier declared within Q. If library unit Q is a package, then STANDARD.Q.x would refer to an identifier declared in package Q, whereas Q.x would refer to an identifier in subunit Q.

S9. A subunit can have the same name as some identifier declared in STANDARD, since subunit names are not implicitly declared in STANDARD.

S10. See IG 8.2/S for a discussion of what it means for one library unit's scope to include another library unit.

Changes from July 1982

S11. There are no significant changes.

Changes from July 1980

S12. Visibility rules take advantage of the fact that library units are implicitly declared in package STANDARD.

S13. PRIORITY and SELECT_ERROR are no longer declared in the predefined package STANDARD, so these can now be names of library units.

Legality Rules

L1. The identifier of a library unit cannot be identical with a predefined identifier declared in STANDARD except that a library unit subprogram can have the identifier TRUE or FALSE if it is not a *parameterless* function with return type STANDARD.BOOLEAN (RM 8.6/2, RM 8.3/15, and AI-00330).

Test Objectives and Design Guidelines

T1 Check that the name STANDARD.M is illegal if a with clause for library unit M is not applicable to the unit being compiled.

Implementation Guideline: Include a case where M is the name of a previously compiled library unit on which the unit being compiled depends.

Check that a library unit cannot have a name identical to a type, exception, or package declared in STANDARD, namely, BOOLEAN, INTEGER, FLOAT, CHARACTER, ASCII, NATURAL, POSITIVE, STRING, DURATION, CONSTRAINT_ERROR, NUMERIC-

`_ERROR`, `PROGRAM_ERROR`, `STORAGE_ERROR`, or `TASKING_ERROR`, plus any implementation-defined numeric type name (e.g., `SHORT_FLOAT`, `LONG_FLOAT`, `SHORT_INTEGER`, and `LONG_INTEGER`).

Check that a library unit can have the name `PRIORITY` or `SELECT_ERROR`.

Check that a library package or generic unit cannot have the name `TRUE` or `FALSE`.

Check that a library subprogram can have the name `TRUE` or `FALSE` if the library unit is not a parameterless function returning type `BOOLEAN`, and cannot have the name `TRUE` or `FALSE` if it is a parameterless function returning type `BOOLEAN` (AI-00330).

- T2. Check that a library unit can be given the name `STANDARD` without replacing the predefined `STANDARD` package.

Implementation Guideline: Try compiling both a `STANDARD` package (with redefinitions of some predefined types and operators) and a subprogram called `STANDARD`. Check that the predefined `STANDARD` package is not replaced by this new unit.

- T3. Check that `STANDARD` is not treated as a reserved word in selected component names.

Implementation Guideline: Declare records and packages with the name `STANDARD` and attempt to reveal errors by using names such as `STANDARD.INTEGER` or `STANDARD.NUMERIC_ERROR` when these names do not refer to the predefined entities.

- T4. Check that the name `STANDARD.M` is legal if a `with` clause for library unit `M` is applicable to the unit being compiled.

Implementation Guideline: Check for all forms of library units and secondary units, including subunits. (Library units can be declared as subprogram declarations, package declarations, subprogram bodies, generic package declarations, generic subprogram declarations, and generic instantiations.)

- T5. Check that a library package, generic unit, or parameterless function returning predefined type `CHARACTER` can have the same name as one of the nongraphic characters, namely, `NUL`, `SOH`, `STX`, `ETX`, `EOT`, `ENQ`, `ACK`, `BEL`, `BS`, `HT`, `LF`, `VT`, `FF`, `CR`, `SO`, `SI`, `DLE`, `DC1`, `DC2`, `DC3`, `DC4`, `NAK`, `SYN`, `ETB`, `CAN`, `EM`, `SUB`, `ESC`, `FS`, `GS`, `RS`, `US`, and `DEL`.

Implementation Guideline: It is not necessary to test all forms of library unit declaration for each name, but each form should be used at least once in this test (see guideline for T1 for forms of library unit declarations).

Check that these identifiers are not directly visible in `STANDARD`.

- T6. Check that each of the required declarations is actually declared in package `STANDARD` and can be accessed by an expanded name, `STANDARD.xxx`.

Implementation Guideline: Include checks for implicitly declared operators.

Implementation Guideline: Check exception names in handlers and raise statements.

- T7. Check that an expanded name for an entity declared in the visible part of a library package can start with the name `STANDARD`.

8.7 The Context of Overload Resolution

Semantic Ramifications

S1. The RM allows certain constructs to be *overloaded*, i.e., have more than one potential interpretation. The process of determining a unique interpretation for an overloaded construct is called *overloading resolution*. An Ada program is illegal unless overloading resolution can determine a unique interpretation for every construct in the program.

S2. This section is divided into two parts. The first part describes overloaded and otherwise ambiguous constructs, and provides a general algorithm for resolving these ambiguities. The second part gives specific rules for resolving potential ambiguities.

8.7.a General Rules for Overloading Resolution

Semantic Ramifications

S1. The RM allows the following constructs to be overloaded: subprograms (RM 6.6/2), single entries of tasks (RM 9.5/5), enumeration literals (RM 3.5.1/5), and basic operations (RM 8.7/1) (in particular, assignment, aggregates, string literals, and allocators). A construct is overloaded if the syntax rules and the visibility rules do not suffice to provide a unique interpretation. The *meaning* of an overloaded occurrence is the declaration of the entity denoted by the occurrence (RM 8.3/1). The declaration may be either implicit or explicit. The *interpretation* of an occurrence determines how the entity contributes to the overall effect of a program. For example, if F is a function returning a record with component X, the interpretation of F in the name F.X could be that F is invoked. But if F.X occurs inside the body of F, F is interpreted as the name of the enclosing unit, and is not invoked (RM 4.1.3/19). Under either interpretation, however, F denotes the same declared entity, i.e., has the same *meaning*.

S2. The following constructs may be overloaded:

- names of user-defined functions and procedures that have identifiers as designators (RM 6.6/2).
- names of user-defined operations that have operator symbols as designators (RM 6.6/2, and RM 6.7/1),
- predefined operators and their operator symbols (RM 6.7/1),
- names of single entries (RM 9.5/2, RM 6.6/2),
- enumeration literals with the form of an identifier (RM 3.5.1/5),
- enumeration literals that are character literals (RM 3.5.1/5),
- basic operations as follows:
 - assignment: for every nonlimited type (RM 7.4.4/1),
 - allocators: for every access type (RM 3.8.2/1),
 - membership tests without ranges: for every type (RM 4.5.2/10),
 - membership tests with ranges: for every scalar type (RM 4.5.2/10),
 - short-circuit control forms: for every boolean type (RM 3.5.5/1),
 - selected component notation: for discriminants (RM 3.7.4/1), record components (RM 3.7.4/1), task single entries (RM 9.5/4), task entry families (RM 9.5/4), and objects designated by access values (RM 3.8.2/2),
 - indexed components: for entries of entry families and for components of arrays (RM 4.1.1/3),
 - slices: for all one-dimensional array types (RM 3.6.2/1),
 - qualification (in a qualified expression): for every type (RM 3.3.3/6),
 - explicit conversion: for all types (see RM 4.6),
 - implicit conversion: for literals, named numbers, and attributes of type *universal_integer* and *universal_real* (RM 3.5.5/1, RM 3.5.8/1, and RM 3.5.10/1),
 - aggregates: for every nonlimited composite type (RM 3.6.2/1 and RM 3.7.4/1),

- language-defined attributes often apply either to all types, or to all types of a certain class (RM Article A),
 - implementation-defined attributes may apply to any class of types (RM 4.1.4/4),
 - names of some language-defined attributes are overloaded, and names of implementation-defined attributes may be overloaded.
- literals as follows:
 - character literals may belong to more than one enumeration type (RM 3.5.1/5),
 - the literal null belongs to every access type (RM 3.8.2/1),
 - string literals may belong to more than one array-of-character type (RM 3.6.2/1).

S3. Numeric literals, named numbers, and attributes that return *universal_integer* or *universal_real* are not really overloaded (because the visibility rules provide a single interpretation); it is their implicit conversion to a nonuniversal type that is overloaded. Since implicit conversions have no notation whatsoever, it is convenient to consider implicitly convertible operands as denoting a potential implicit conversion. Thus, although the name of a named number is not overloadable, there are multiple interpretations of a named number as a primary; the set of potential interpretations must include every possible implicit conversion as well as the possibility of no implicit conversion.

S4. In many cases, the syntax and visibility rules alone are sufficient to determine that a particular occurrence denotes a particular declaration, or that a particular occurrence is illegal because no applicable declaration exists. In some cases (listed above) more than one declaration might be denoted, or more than one interpretation might apply. Overloading resolution rules are used to disambiguate overloaded notations.

S5. Overloading resolution is performed at compile time. Every overloaded constituent of every program unit must be given a unique interpretation whether or not it will ever be evaluated. For example, constituents of generic units must be resolved whether or not the unit is ever instantiated. This resolution uses the declarations visible inside the generic declaration and body. An instance uses the resolutions determined when the unit was compiled; the only overloading resolution performed at an instantiation is for generic actual parameters (including default subprograms specified with a box, but excluding all other default parameters) (see IG 12.3.6/S).

8.7.a.1 Nonoverloadable Constructs

Semantic Ramifications

S1. The visibility rules only allow overloading for operations; therefore, these rules ensure there is a unique interpretation for nonoverloadable constructs. In particular, simple names denoting declared objects (including named numbers, constants, variables, discriminants and other record components, formal parameters, loop parameters, and deferred constants), types, subtypes, packages, tasks, exceptions, entry families, loops, blocks, and statement labels are not overloadable. Names of generic units are not overloadable except inside generic subprograms, where the name of the generic unit is the name of a subprogram.

S2. Although certain simple names are not overloadable, expanded names can contain prefixes with overloaded names:

```

type R is
  record
    X : INTEGER;
  end record;
function F return R is ... end F;
function F return INTEGER is
  X : INTEGER := 0;
  Y : INTEGER := F.X;

```

The name, F.X, contains a prefix, F, that is overloaded since two declarations of F are visible. The overloading resolution process determines that in the above case, F denotes the enclosing function and is not invoked (hence, F.X denotes the variable X, not a record component). So although a simple name denoting a named number, object declaration, formal parameter, loop parameter, deferred constant, type, subtype, package, task unit, entry family, exception, generic unit, loop, block, and statement label cannot be overloaded, an expanded name denoting such an entity can contain a prefix having an overloaded constituent.

S3. Prefixes of selected components using rules (a) through (d) of RM 4.1.3 are also overloadable, since these prefixes may be function calls. Similarly, the prefixes of indexed components and slices may be overloaded (see RM 4.1), and the indexed component basic operations are themselves overloaded. Therefore, constituents of prefixes of names denoting objects (including task objects), entries, and entry families may be overloaded. Constituents of prefixes of attributes may be overloadable, depending on the attribute (see RM 4.1.4, RM 4.1, and RM A).

8.7.a.2 Syntactic Ambiguity

Semantic Ramifications

S1. Overloading resolution depends partly on the syntactic structure of an Ada program. For example, in a conditional entry call:

```

select
  E;
else
  null;
end select;

```

E can only be parsed as an entry_call_statement. This implies that the only possible interpretations of E are as an entry call. For example:

```

task T is
  entry E;
end T;

task body T is
  procedure E (X : INTEGER := 0) is
  begin ... end E;
begin
  select
    E;          -- unambiguous; entry E
  else
    null;
  end select;
  E;          -- ambiguous

```


The second call to E is ambiguous because it can be parsed as either a procedure call statement or as an entry call statement, and there exist a visible procedure E and a visible entry E that can be called with no parameters (RM 8.7/13). In the first call, the only syntactic alternative is *entry_call_statement*, and this suffices to resolve which declaration is denoted by this occurrence of E.

S2. The RM's context-free syntax rules define the syntactic structure that affects overloading resolution. However, the context-free syntax given in the RM is ambiguous in ways that interact with overloading resolution, at least conceptually. In practice, it is not important that overloading resolution helps to resolve syntactic ambiguities that occur in the RM grammar. An actual implementation will use an unambiguous, context-free grammar such as that given in Volume II, Number 2, of *Ada Letters*.

S3. For example, consider the name F(X) when the following declarations of F are visible:

```
type ARR is array (1..2) of INTEGER;
function F return ARR;                -- F1
function F (Y : INTEGER) return CHARACTER; -- F2
```

Given these declarations, F(X) can be parsed as either an indexed component whose prefix is the parameterless function call F, or as a function call with actual parameter X. The choice of syntactic structure, and the interpretation of F, is determined by the context in which F(X) appears, i.e., overloading resolution determines the correct parse:

```
Y : INTEGER := F(1);    -- calls F1 and indexes return value
Z : CHARACTER := F(1);  -- calls F2
```

In the grammar given in *Ada Letters*, F(X) is parsed as an indexed component, and later analysis determines whether the "indexed component" is really a function call or not.

Note that $F(1) = F(1)$, however, is ambiguous because equality is visible for both type INTEGER and type CHARACTER, and so it cannot be determined whether F(1) is an indexed component or a function call. However, if we write:

```
F(Y => 1) = F(1)
```

there is no ambiguity, because F(Y => 1) can only be parsed as a function call, and hence yields a result of type CHARACTER.

S4. There are only a few situations in which the syntax rules play a critical role in overloading resolution. These cases are:

- the parse of a conditional entry call or a timed entry call serves to distinguish between calls to an entry and calls to a procedure or function.
- the parse of a procedure call statement or an entry call statement eliminates the possibility of a call to a function, e.g., consider the call P;. Even if there is a visible function P that can be called without any actual parameters, such a function is not a possible interpretation of P in the above statement. On the other hand, if P is overloaded between an entry call and a procedure call, the above context is insufficient to resolve the call if both the entry and the procedure can be called without parameters.
- the parse of a name as a function call eliminates possible interpretations as a procedure call or an entry call.
- the parse of an actual parameter as an aggregate, a string literal, or an allocator can affect overloading resolution (see examples below).

- the use of a named parameter association syntactically distinguishes a function call from an indexed component whose prefix is a parameterless function call.
- a slice can always be distinguished from a function call or an indexed component by using syntax and visibility rules: $F(X..Y)$ is syntactically a slice; $F(Z)$ is a slice if and only if Z denotes a type mark (which is determined by the visibility rules); and $F(T'RANGE)$ is a slice (T must denote a type mark or array).
- the prefix of the attribute $A'ADDRESS$ and $A'SIZE$ cannot be a function call (since the prefix cannot be a value; RM 13.7.2/2, /6), and hence the interpretation of $F'ADDRESS$ is syntactically unambiguous (and completely unambiguous if there is only one visible F).

S5. There are cases where the context-free syntax of the RM is insufficient to disambiguate constructs that seem unambiguous to a programmer. In these cases, context-dependent syntax rules are used to resolve the ambiguity. In particular, consider the following:

```

type ARR is array (1..2) of CHARACTER;
procedure P (X : ARR);           -- P1
procedure P (X : CHARACTER);     -- P2
...
P (('A', 'B'));                  -- unambiguous; calls P1
P (('C'));                       -- unambiguous; calls P2

```

The first call is unambiguous because ('A', 'B') can only be parsed as an aggregate; hence, the only interpretations of P that can be considered are those that require a composite type as P 's first argument. This eliminates $P2$ as a possibility. In the second call, $P(('C'))$, the context-free syntax of the RM allows ('C') to be parsed as either a parenthesized expression or as an aggregate. There is, however, a context-dependent rule that disallows one-component positional aggregates (RM 4.3/4); this rule is used to eliminate the parse of ('C') as an aggregate. Note that if ('C') were considered an aggregate, then all P s taking composite and scalar types as first arguments must be considered possible interpretations of P , and the second call to P could not be resolved.

S6. Since context-dependent rules are used only to resolve ambiguities in the RM's context-free syntax, structures that may seem unambiguous to a programmer are not always resolved by the RM's syntax. For example, the form of a choice in an aggregate cannot be used to resolve a call. If we take the preceding example and add the following declarations:

```

type REC is
  record
    A, B : CHARACTER;
  end record;
procedure P (X : REC);

```

Then the call

```
P (1..2 => 'A');
```

is ambiguous, since $(1..2 => 'A')$ is parsed simply as an aggregate; the fact that it can only legally be an array aggregate because of the form of the choice is not taken into consideration in the RM's syntactic structure. Hence, the syntactic structure provides no useful information for overloading resolution. However, the call

```
P ("AB");
```

is unambiguous, because "AB" is syntactically a string literal, i.e., a one-dimensional array of a character type, and this information can be used to resolve the call (RM 8.7/12 and RM 4.2/4).

S7. The visibility rules are also used to resolve ambiguities in the context-free syntax. For example, given a declaration of the form:

```
X : T(F);
```

the structure, (F), is an index constraint if F denotes a type mark and is otherwise a discriminant constraint. Similarly, U(1) is potentially an indexed component, a function call, or a type conversion. If U is the name of an array object or type, the visibility rules uniquely determine that U is an indexed component or a type conversion, respectively, since array and type names cannot be overloaded.

S8. After using the context-free, context-dependent, and visibility rules to determine a syntactic structure, there are still some syntactic ambiguities resolved only during the overloading resolution process. This interaction between overloading resolution and the determination of a syntactic structure is of no practical consequence as long as an implementation does not use the RM's context-free syntax as its implementation syntax. Resolving these syntactic ambiguities provides no additional information beyond that produced by the resolution process itself. For example, the name F appearing as an actual generic parameter can be parsed as either a function call or as the name of a function; the decision is determined by the nature of the corresponding formal parameter. If the formal parameter is a formal subprogram, then F is parsed as a name; if the formal parameter is an object, then F is parsed as a function call:

```
generic
  X : INTEGER;
  with function FF return INTEGER;
package P ... end P;

package NP is new P(F, F);
```

The first parameter is parsed as a function call; the second is not. The resolution of this syntactic ambiguity is secondary to deciding which function is denoted by each occurrence of F. An actual implementation could parse both occurrences of F as names, and could later decide whether the interpretation of a particular name is that it is a parameterless function call.

S9. There are some cases where syntactic information is ignored during the resolution process. For example, consider:

```
procedure P (X : INTEGER);
procedure P (X : in out INTEGER;
             Y : INTEGER := 0);

P(3):      -- ambiguous
```

Here the parse of P(3) unambiguously indicates that the actual parameter is an expression, not a variable name. The visibility rules say that both declarations of procedure P are visible. The overloading resolution process ignores the mode of formal parameters in resolving a subprogram name (RM 8.7/13 and RM 6.6/3). Hence, the syntactic information is not used, and the call is considered ambiguous.

S10. In short, the context-dependent and visibility rules are used to resolve ambiguities in the context-free parse of an Ada program. To the extent that these rules eliminate syntactic alternatives, the resulting syntactic structures (sometimes more than one) provide information that can be used by the overloading resolution process, but not all syntactic information is necessarily used.

S11. The following is a list of the contexts in which the visibility rules and the syntax rules (both context-free and context-dependent) do not suffice to determine a unique interpretation of an overloaded name:

- Aggregate choices that may be either simple names of components or simple expressions, e.g., for $(F \Rightarrow 1)$, is F a parameterless function call or the name of a record component?
- Parameterless function calls vs. simple names in prefixes of selected components, e.g., for $F.C$, is F a parameterless function call or the name of an enclosing unit (in which case, $F.C$ is an expanded name and F is not invoked)?
- Parameterless function calls vs. simple names as generic actual parameters, e.g., F as an actual generic parameter is a call if the formal parameter is an object and it is a name if the formal parameter is a subprogram.
- Positional function calls with parameters vs. indexed components, e.g., is $F(X)$ a function call or an indexed array? (Note: $F(X)$ is syntactically a slice if X is the name of a discrete type (or is the range attribute, $Y'RANGE$); this interpretation of $F(X)$ is resolved by the visibility rules. Also, $F(L \Rightarrow X)$ is unambiguously a function call according to the context-free syntax.)
- String literals vs. operator symbols as generic actual parameters, e.g., $+$ is the name of a subprogram if the formal parameter is a subprogram and it is a string literal if the formal parameter is an object.

Each of the above potential ambiguities must be resolved by the overloading resolution process. The sets of resolutions constructed during the bottom-up phase of the algorithm given below must consider all possible syntactic interpretations of the above ambiguous constructs in parallel. Note that a parameterless function call need not call a parameterless function, if each parameter has a default expression.

8.7.a.3 Ambiguities Regarding Visibility

Semantic Ramifications

S1. Although the visibility rules are sometimes used to resolve syntactic ambiguities, thereby eliminating certain possibilities for overloading resolution, overloading resolution also determines which identifiers are visible in certain contexts. In particular,

- In a selected component, the visibility of the selector depends on the resolution of the prefix.
- In a subprogram call or an entry call with named parameter associations, the visibility of the simple names of formal parameters depends on the resolution of the name of the subprogram or the entry.
- In a record aggregate with named component associations, the visibility of component simple names depends on the resolution of the aggregate.

The specific overloading resolution rules listed in IG 8.7.b/S determine how the visibility rules are applied during the overloading resolution process.

S2. In addition, within a task body an occurrence of the name of a task unit may denote either the task unit or the task (value) executing the body (RM 9.1/4 and RM 4.1.3(f)). In this case, the name of the unit is always interpreted as the value of the innermost task object executing the statements of the task body unless the name is used as a prefix of an expanded name; in that case, the name denotes the unit.

S3. Note that a task name is not overloaded in the sense that such a name can be associated with more than one declaration. The only ambiguity involving a task name is its interpretation as a task object or as the name of an enclosing unit. Resolving this interpretation is not a part of overloading resolution as conceived and defined by the RM, since task names are not overloaded. But we consider it part of overloading resolution in our discussion here.

8.7.a.4 Complete Contexts for Overloading Resolution

Semantic Ramifications

S1. It can be shown that every overloaded occurrence occurs inside one of the following constructs (the difference between expression and simple expression is ignored here):

- Object declaration (in a range constraint, an index constraint, a discriminant constraint, an initial expression, or a range in the index constraint of a constrained array definition) (see RM 3.2 and RM 3.6).
- Number declaration (in an expression) (see RM 3.2).
- Subtype declaration (in a subtype indication) (see RM 3.3.2).
- Derived type definition (in a subtype indication) (see RM 3.4).
- Integer or real type definition (in an expression of a range or accuracy definition) (see RM 3.5.4, RM 3.5.6, RM 3.5.7, and RM 3.5.9).
- Array type definition (in either a subtype indication or a range in the index constraint of a constrained array definition) (see RM 3.6).
- Record component declaration (in a subtype indication or a default expression) (see RM 3.7).
- Discriminant specification (in a default expression) (see RM 3.7.1).
- Record type definition (in a choice in a variant) (see RM 3.7.3 and RM 3.7).
- Access type definition (in a subtype indication) (see RM 3.8).
- Prefix of an attribute (see RM 4.1.4).
- Operand of a type conversion (see RM 4.6).
- Entry declaration (in the discrete range of an entry family) (see RM 9.5).
- Parameter specification (in a default expression) (see RM 6.2).
- Generic parameter declaration (in the default expression for an object or a subprogram parameter) (see RM 12.1 and RM 12.1.2).
- Renaming declaration (in an object name, a subprogram name, or an entry name) (see RM 8.5).
- Generic instantiation (in a generic actual parameter that is an expression, an object name, a subprogram name, or an entry name) (see RM 12.3).
- Representation clause (in an expression, an array aggregate, or a range) (see RM 13.1, RM 13.2, RM 13.3, RM 13.4, and RM 13.5).
- Assignment statement (both sides) (see RM 5.2).
- If statement (in a condition) (see RM 5.3).
- Case statement expression (see RM 5.4 and RM 8.7).

- Case statement (in a choice) (see RM 5.4).
- Loop parameter specification (in the discrete range) (see RM 5.5).
- Loop statement (in a while condition) (see RM 5.5).
- Exit statement (in a condition) (see RM 5.7).
- Return statement (in the expression) (see RM 5.8).
- Procedure or entry call (in an actual parameter or in the name of the procedure or entry) (see RM 6.4 and RM 9.5).
- Accept statement (in an entry simple name or an entry family index expression) (see RM 9.5).
- Delay statement (in the expression) (see RM 9.6).
- Select statement (in a condition of a selective wait) (see RM 9.7.1).
- Abort statement (in a task name) (see RM 9.10).
- Code statement (in the aggregate) (see RM 13.8).
- Type conversion operand (in an explicit type conversion primary or actual subprogram or entry parameter of mode in out or out) (see RM 4.6, RM 6.4, RM 6.4.1, and RM 8.7).

S2. The constructs listed above serve as "complete contexts" for overloading resolution of constituents. If more than one such construct encloses an overloaded occurrence, the innermost construct defines the "complete context." The information for overloading resolution is supplied by the complete context, the visibility rules, and the syntactic structure. Specific overloading resolution rules (in IG 8.7.b/S) define exactly what contextual information is used. In general, the entire context must be used to resolve any constituent, so overloading resolution of a constituent may depend on other constituents that occur far away (they must occur in the same complete context). The only information provided from outside the complete context is the set of possible meanings provided by the visibility rules and by the syntactic structure.

S3. Since a statement is a complete context, the context for overloading resolution of entry call statements is identical to that of procedure call statements, except in a conditional or timed entry call. In such statements, the syntactic structure indicates that the statement following select must be an entry call statement, and this syntactic information is used to eliminate possible resolutions as procedure calls. Hence, procedures and renamings of entries are not considered as possibilities in this syntactic context.

8.7.a.5 A Model for an Overloading Resolution Algorithm

Semantic Ramifications

S1. Overloading resolution can be performed by a bottom-up traversal of the syntax tree followed by a top-down traversal (a syntax tree is sometimes called a dependency tree or a parse tree). The bottom-up phase first propagates information about the constituents of constructs to resolve enclosing constructs. The top-down traversal then propagates information from an enclosing construct (the "context") in order to completely resolve each constituent construct. This model is purely conceptual; an implementation may use a different (but equivalent) algorithm.

S2. A construct is a node in the syntax tree. A syntax rule may contain embedded constructs.

here called constituents. The constituents are the children of the constructs in a syntax tree. The root of the syntax tree is a compilation (see RM 10.1). Irrelevant syntax rules will be ignored throughout the following discussion.

S3. The information propagated between levels of the syntax tree takes the form of *resolutions*. The resolution of a construct is defined in the table below. For example, resolutions of the expression $X + Y$ are specified in terms of the type of the result of "+". Similarly, resolutions of the primary X are the possible types of X . Resolutions of the simple_name, X , however, are the various visible declarations of X . One goal of the resolution process is to associate each simple name, operator symbol, operator, literal, or basic operation in a complete context with a single declaration. If no such unique association can be found, the construct is illegal.

Syntactic category:		Resolution:
Primary	=>	Declaration of the type of the result value
Expression	=>	Declaration of the type of the result value
Relation	=>	Declaration of the type of the result value
Simple expression	=>	Declaration of the type of the result value
Term	=>	Declaration of the type of the result value
Factor	=>	Declaration of the type of the result value
Simple name	=>	Declaration of the simple name
Character literal	=>	Declaration of the character literal
Operator symbol	=>	Declaration of the operator symbol
Operator	=>	Declaration of the operator symbol
Assignment	=>	Declaration of the basic operation
Allocator	=>	Declaration of the basic operation
in, not in	=>	Declaration of the basic operation
and then, or else	=>	Declaration of the basic operation
Selected component	=>	Declaration of the basic operation
Indexed component	=>	Declaration of the basic operation
Slice	=>	Declaration of the basic operation
Attribute	=>	Declaration of the result type
Qualified expression	=>	Declaration of the basic operation
Type conversion	=>	Declaration of the basic operation
type_mark(name)	=>	Declaration of the conversion operation
Numeric literal	=>	Declaration of the literal's type declaration of implicit conversion's result type (see below)
null	=>	Declaration of access type
String literal	=>	Declaration of array of character type
Aggregate	=>	Declaration of the basic operation
Choice	=>	Declaration of the type of the choice
Actual subprogram or entry parameter	=>	Declaration of the type of the parameter
Prefixes:		
Function call	=>	Declaration of the type of the result value
Object name	=>	Declaration of the base type of the object
Entry of family	=>	Declaration of the entry family
Other prefix	=>	Declaration of the name or of the attribute denoted by name

8.7.a.6 An Algorithm for Overloading Resolution

Semantic Ramifications

S1. The following two-phase algorithm can be used to determine the interpretation of a complete context by propagating sets of resolutions and matching them against potential interpretations. The steps are performed in the following order:

1. The bottom-up phase constructs a set of possible resolutions for each construct. The set of possible resolutions for a construct depends only on the syntactic structure, the visibility rules, the sets of resolutions of constituent constructs, the specific overloading resolution rules listed in IG 8.7.b/S, and a special rule for selecting resolutions requiring the fewest implicit conversions of *universal_real* or *universal_integer* values (see later discussion). Each set of resolutions is saved until the top-down phase (when a unique resolution is selected, if one exists).
2. Upon reaching a complete context (i.e., a statement, a declaration, or a representation clause), the rules for complete contexts (also in IG 8.7.b/S) are applied to eliminate possible resolutions of constituents that are incompatible with the complete context or with each other. If there is no unique resolution in the complete context, then the program is illegal.
3. The top-down phase provides each construct with a unique resolution, if possible. All meanings of a construct that are incompatible with this required resolution are discarded. The program is illegal unless the construct has exactly one resolution. This resolution determines a unique required resolution for each constituent of the construct. Each resolution is propagated downwards to the next lower level in the syntax tree. In this manner information from the context is used to determine the actual interpretation of each construct from a set of possible interpretations. This completes the algorithm.

S2. The set of resolutions constructed during the bottom-up phase must comprise exactly the resolutions allowed by the specific overloading resolution rules listed in IG 8.7.b/S. This set must include separate resolutions for every syntactic or semantic ambiguity, and must consider every combination of resolutions of constituents of each construct. For each operator, and for each name that is a simple name, an operator symbol, or a character literal, every directly visible declaration must be considered.

S3. For example, the statement `F.all := G;` is resolved as follows, given that only the following declarations of `F` and `G` are visible:

```
package P is
  type LP is limited private;
  ...
end P;
use P;

type ALP is access LP;
type REC is
  record X : CHARACTER; end record;
type AS is access STRING;
type AF is access FLOAT;

function F return AS;                                -- F1
```



```

function F return INTEGER;           -- F2
function F return AF;                -- F3
function F return REC;               -- F4
function F return ALP;               -- F5
function F (X : INTEGER) return FLOAT; -- F6

function G return STRING;            -- G1
function G return INTEGER;           -- G2

```

S4. The bottom-up phase begins by constructing the resolution sets for the simple name F and for the simple name G. All directly visible F's and G's are included. We will use the notation:

```

FLOAT <- F(INTEGER)

```

to distinguish between overloadings of a particular operation. This notation indicates that result type FLOAT is returned by the visible F operation that requires an operand of type INTEGER. For each resolution set, we will indicate the syntactic category associated with the set and the elements of the set:

```

simple_name; F : S1 = {AS      <- F,
                      INTEGER <- F,
                      AF      <- F,
                      REC      <- F,
                      ALP      <- F,
                      FLOAT    <- F(INTEGER) }

simple_name; G : S2 = {STRING  <- G,
                      INTEGER <- G}

```

S5. The resolution set for F considered as a function call is obtained by attempting to apply each operation in S1. We indicate this by the notation $S_x(S_y, S_z)$, where S_x is the resolution set for an operation, and S_y and S_z are the resolution sets for the parameters of the operation. We indicate the source of the operation and the operands by appropriate subscription, e.g., $F[S1.1]$ indicates the F that is the first element in set S1 (i.e., the resolution that F denotes F1). Since all the functions in S1 except S1.6 can be invoked without any actual parameters, and since RM 8.7/13 and RM 6.6/3 allow us to consider the number of actual parameters in resolving a call, we can exclude S1.6 as a possible interpretation of F when F is considered as a function call. Hence, the result type FLOAT does not enter into the resolution set for F considered as a function call.

```

call; S1 : S3 = {AS      <- F[S1.1],
                 INTEGER <- F[S1.2],
                 AF      <- F[S1.3],
                 REC      <- F[S1.4],
                 ALP      <- F[S1.5] }

call; S2 : S4 = {STRING  <- G[S2.1],
                 INTEGER <- G[S2.2] }

```

S6. The next syntactic construct to be considered is the selected component, F.all. We first consider what component selection operations are visible:

```

selection; . : S5 = {STRING <- select (AS, all),
                    FLOAT  <- select (AF, all),
                    INTEGER <- select (REC, X),
                    LP      <- select (ALP, all) }

```

The notation "STRING <- select(AS, all)" means that there is a component selection operation declared for a prefix of type AS and a selector having the form all. This component selection operation returns a result of type STRING. By considering the form of the name F.all and the visible component selection operations, we can determine the resolution set for F.all. In particular, since the only select operation for type REC does not allow a selector of the form .all, resolution of F.all to type INTEGER is not possible. Similarly, since no selector operation is defined for F2's result type, the resolution S3.2 is not further considered. Hence, the resolution set for F.all is:

```
selected component; select(S3, all) :
    S6 = {STRING <- select[S5.1] (AS[S3.1], all),
          FLOAT  <- select[S5.2] (AF[S3.3], all),
          LP     <- select[S5.4] (ALP[S3.5], all)}
```

S7. The next relevant node in the syntax tree is the assignment statement itself. We first consider all visible assignment operations:

```
operation; := : S7 = {:= (STRING, STRING),
                       := (INTEGER, INTEGER),
                       := (FLOAT, FLOAT),
                       ...}
```

Note that assignment is not visible for type LP. Given the resolution sets for F.all and the function call G (namely, sets S6 and S4), and the requirements of the visible assignment operations, the only possible resolution for := is the resolution that takes arguments of type STRING:

```
assignment statement; :=(S6, S4) :
    S8 = {:= [S7.1] (STRING[S6.1], STRING[S4.1])}
```

Since the assignment statement is a complete context, the bottom-up phase stops propagating information at this point.

S8. The top-down phase begins by checking that there is exactly one resolution in the resolution set for the complete context. This resolution determines the required resolutions of the immediate constituents of the assignment statement; hence, resolutions S4.2, S6.2, and S6.3 are discarded from sets S4 and S6. Since this leaves both sets S4 and S6 with unique resolutions, the top-down propagation of information can continue. Since the only resolution now present in S6 was derived from S3.1, all resolutions in S3 except the first are discarded. Similarly, only S2.1 is retained. S2 and S3 now contain unique resolutions. The resolution in S3 was derived from S1.1, so all other resolutions in S1 are discarded, and the process has determined a unique interpretation for F, G, component selection, and :=.

8.7.a.7 Implicit Conversions of Numeric Types

Semantic Ramifications

S1. A numeric literal, named number, or attribute returning type *universal_integer* or *universal_real* can be implicitly converted to an integer or real type, respectively. Implicit conversions are applied if, and only if, there is no legal interpretation of an expression without the implicit conversion (RM 4.6/16). This rule is a preference rule. It says that when an ambiguity arises due to implicit conversions, the implicit conversions are not to be performed.

S2. As a simple example, consider the following:

```
if 5 = A'LENGTH then ...
```

Both 5 and A'LENGTH have type *universal_integer* (RM 2.4/1 and RM 3.6.2/10). An implicit conversion to INTEGER can be applied to both operands. If the conversion is applied, then the resolution of "=" can be either the equality operator taking arguments of type INTEGER or the operator taking operands of type *universal_integer*. Because of the preference rule given in RM 4.6/16, this potential ambiguity is resolved in favor of the interpretation not requiring an implicit conversion.

S3. In terms of the overloading resolution algorithm, the bottom-up resolution step must take into account the possibility of implicit conversions and must record for later use whether any implicit conversions have been performed lower in the syntax tree. For example, consider the following resolutions for 5 = A'LENGTH (where UI stands for *universal_integer* and "conv" represents the operation for converting values of type *universal_integer*):

```
literal: 5 :      S1 = {UI,
                    INTEGER:1 <- conv(UI)}

primary: A'LENGTH : S2 = {UI,
                          INTEGER:1 <- conv(UI)}
```

Here we assume that the only nonuniversal integer type whose scope includes the expression is the type INTEGER. We use the notation INTEGER:1 to indicate that the result type INTEGER has been obtained by an implicit conversion. We will see later that it is necessary to keep track of how many implicit conversions have been performed, so the :1 indicates that a single conversion was needed to produce the result type INTEGER.

S4. S3 specifies the set of visible equality operators:

```
operator: "=" : S3 = {BOOLEAN <- "="(UI, UI),
                      BOOLEAN <- "="(INTEGER, INTEGER),
                      BOOLEAN <- "="(FLOAT, FLOAT),
                      ...}
```

The resolution set for the relation 5 = A'LENGTH is obtained by attempting to apply each operation in S3 to the operands in S1 and S2, respectively.

```
relation: S3(S1, S2) :
  S4 = {BOOLEAN <-
        "="[S3.1] (UI[S1.1], UI[S2.1]),
        BOOLEAN:2 <-
        "="[S3.2] (INTEGER:1[S1.2], INTEGER:1[S2.2])}
```

Note that application of "="[S3.2] produces a BOOLEAN result, but requires two implicit conversions in the underlying syntax tree. Since both applicable operators in S3 produce the same result type, and since one of the operators requires more implicit conversions than the other, the second resolution in S4 can be discarded at this point; S4 really contains just the result of applying S3.1.

S5. Since the if statement is a complete context, and since the condition of an if statement requires a boolean type, and since S4 contains a unique boolean type, the relation can be completely resolved, and so is unambiguous. It is only unambiguous, however, because the preference rule in RM 4.6 allows S4.2 to be discarded.

S6. The reason for keeping track of the number of implicit conversions is illustrated by the following example (inspired by an example originally given by Peter Belmont). Consider the following declarations:

```

function F (X : INTEGER) return INTEGER;
function F (X : BOOLEAN) return INTEGER;
function F (X : INTEGER) return FLOAT;

function "<" (L, R : INTEGER) return INTEGER;
...
X : INTEGER := F(3**4 < 5);
Y : FLOAT   := F(3**4 < 5);

```

S7. Now let us consider the resolution sets produced for 3**4:

```

primary: 3 : S1 = {UI,
                  INTEGER:1 <- conv(UI)}

primary: 4 : S2 = {UI,
                  INTEGER:1 <- conv(UI)}

operator: "***" : S3 = {UI      <- "***" (UI, INTEGER),
                       INTEGER <- "***" (INTEGER, INTEGER)}

```

Note that the second operand of the exponentiation operator must have type INTEGER (RM 4.5.6/5), so we obtain the following resolution set for the factor, 3**4:

```

factor: S3(S1,S2) :
  S4 = {UI:1 <-
        "***"[S3.1] (UI[S1.1], INTEGER:1[S2.1]),
        INTEGER:2 <-
        "***"[S3.2] (INTEGER:1[S1.2], INTEGER:1[S2.2])}

```

Note that two implicit conversions are required to achieve the result type INTEGER, and even the result type *universal_integer* can only be achieved by applying an implicit conversion to the second operand of "**". We now proceed to develop the resolution set for the relation 3**4 < 5:

```

primary: 5 : S5 = {UI,
                  INTEGER:1 <- conv(UI)}

operator: "<" : S6 = {BOOLEAN <- "<"(UI, UI),
                    BOOLEAN <- "<"(INTEGER, INTEGER),
                    INTEGER <- "<"(INTEGER, INTEGER),
                    ... }

relation: S6(S4,S5) :
  S7 = {BOOLEAN:1 <-
        "<"[S6.1] (UI:1[S4.1], UI[S5.1]),
        INTEGER:3 <-
        "<"[S6.3] (INTEGER:2[S4.2], INTEGER:1[S5.2])}

```

Note that application of S6.2 would produce the resolution:

```

BOOLEAN:3 <- "<"[S6.2] (INTEGER:2[S4.2], INTEGER:1[S5.2])

```

but since S7 already contains the result type BOOLEAN, and this result type is obtained with fewer implicit conversions, the BOOLEAN:3 resolution can be discarded immediately.

S8. We now proceed to resolve the function call:

```

name; F : S8 = {INTEGER <- F(BOOLEAN),
               INTEGER <- F(INTEGER),
               FLOAT   <- F(INTEGER)}

call; S8(S7) : S9

```

Since S9.1 and S9.2 have the same result type, but S9.1 requires fewer implicit conversions in its subtree, S9.2 is discarded from the resolution set. Hence, the final version of S9 only contains the result types INTEGER:1 and FLOAT:3. The context of the call to F determines which result type is appropriate. Hence, in the initialization of X, F[S8.1] will be chosen as the resolution of F, whereas in the initialization of Y, F[S8.3] will be chosen. Note in particular that the FLOAT:3 resolution cannot be discarded when forming S9, simply because it requires more implicit conversions than the INTEGER:1 resolution.

S9. Another case requiring special attention is the use of a range in a constrained array definition, an iteration rule, or a declaration of an entry family. In this case, if each bound is either a numeric literal, a named number, or an attribute, and the type of both bounds prior to implicit conversion is *universal_integer*, then an implicit conversion to INTEGER is assumed (RM 3.6.1/2). The effect of this rule is shown by the following example:

```
for I in INTEGER'POS(F(3**4 < 5)) .. 6
```

S10. Using the resolution sets developed above for the analysis of the function call, we arrive at the following resolution sets:

```

attribute; INTEGER'POS : S10 = {UI <- 'POS(INTEGER)}

primary; S10(S9) : S11 = {UI:1 <- 'POS[S10.1] (INTEGER:1[S9.1]),
                        INTEGER:2 <- conv(UI:1[S11.1])}

```

Note: S11.2 represents the implicit conversion from the *universal_integer* result type delivered by the 'POS attribute.

```

primary; 6 : S12 = {UI,
                   INTEGER:1 <- conv(UI)}

```

S11. Although a range is not defined as an operation by the RM, it is convenient for purposes of this discussion to treat .. as an operation taking two arguments of the same scalar type (RM 3.5/4) or two arguments that are both integer types or both real types (RM 3.5.4/3, RM 3.5.7/3, and RM 3.5.9/3):

```

operation; .. : S13 = {..(UI, UI),
                      ..(INTEGER, INTEGER),
                      ..(FLOAT, FLOAT),
                      ..(UI, INTEGER),
                      ..(INTEGER, UI),
                      ...}

range; S13(S11, S12) :
    S14 = {..[S13.1] (UI[S11.1], UI[S12.1]),
          ..[S13.2] (INTEGER:2[S11.2], INTEGER:1[S12.2]),
          ..[S13.3] (UI[S11.1], INTEGER:1[S12.2]),
          ..[S13.4] (INTEGER:2[S11.2], UI[S12.1])}

```

S12. S14 is the complete context. If there were no special rule, RM 4.6/15 would require that S14.1 be chosen as the interpretation of the range, since it requires the fewest implicit conversions. But RM 3.6.1/2 specifies, in essence, that when a range is a complete context, its bounds cannot have type *universal_integer*. Hence, S14.1 is ruled out of consideration, leaving S14.2 as the only interpretation. If LONG_INTEGER were also defined in STANDARD, then

```
LONG_INT:2 <- conv(UI:1[S11.1])
```

would be added to S11, S12 would be

```
S12 = {UI,
      INTEGER:1 <- conv(UI),
      LONG_INT:1 <- conv(UI)}
```

S13 would include the following elements:

```
S13 = {..(UI, UI),
      ..(INTEGER, INTEGER),
      ..(LONG_INT, LONG_INT),
      ..(INTEGER, LONG_INT),
      ..(LONG_INT, INTEGER),
      ...}
```

and the resolution set for S14 would have been:

```
S14 = {..[S13.1] (UI:1[S11.1], UI[S12.1]),
      ..[S13.2] (INTEGER:2[S11.2], INTEGER:1[S12.2]),
      ..[S13.3] (LONG_INT:2[S11.3], LONG_INT:1[S12.3]),
      ..[S13.4] (INTEGER:2[S11.2], LONG_INT:1[S12.3]),
      ...}
```

RM 3.6.1/2 requires us to discard S14.1. If we now just used the rule that both bounds must be the same discrete type (RM 3.6.1/2), S14 would contain two elements, S14.2 and S14.3, and the range would be ambiguous. But RM 3.6.1/2 also specifies that when the bounds have a particular syntactic form and can be resolved to type *universal_integer* (see S11.1 and S12.1), then only resolutions for ..(INTEGER, INTEGER) are to be considered. Since the bounds in the example have the required form and can be resolved to have type *universal_integer*, this rule allows us to discard S14.3 from the set, giving the range an unambiguous interpretation.

S13. If the lower bound had the form, -1, and LONG_INTEGER or some other integer type had been declared in STANDARD, the range would be illegal. The resolution set for -1 would be:

```
-1 : S15 = {UI          <- -(UI),
            INTEGER:1    <- -(INTEGER:1),
            LONG_INT:1    <- -(LONG_INT:1)}
```

In this case, although -1 has *universal_integer* as a possible result type, -1 is not one of the forms specified by the rule in RM 3.6.1/2 (since -1 is not a literal, but is an expression) but S14 has the same members as before. Since we can no longer discard S14.3, the range -1..6 is illegal. If, however, the only visible unary minus operators are those used in S15.1 and S15.2 (e.g., because LONG_INTEGER is not declared in STANDARD), then -1..6 is legal because S14.1 is discarded and S14.2 is the only remaining resolution.

8.7.b Specific Overloading Resolution Rules

Semantic Ramifications

S1. During the bottom-up phase of overloading resolution, the resolutions of constructs are those specified by the following rules. The bottom-up phase starts with the leaves of the syntax tree whose resolutions are determined solely by the visibility rules. For higher levels of the syntax tree, the set of resolutions is constructed from all possible combinations of resolutions of constituents, eliminating resolutions that do not conform to these rules.

S2. The leaves of the syntax tree consist of numeric literals, string literals, character literals, operators and operator symbols, the literal null, the reserved word *all*, names that are simple names, and simple names in accept statements and address clauses. Other constructs (such as parentheses and other punctuation) may be considered leaves, but have no effect on the algorithm.

S3. *Universal_integer* is included among the integer types, and *universal_real* is considered a real type, although it is neither a fixed nor a floating point type. In this description, as in the RM, note that the name of a type may denote either a private type or some other type, depending on the location of the occurrence (and depending on the scope rules). Similarly, the interpretation of a resolution that is a type declaration may depend on the scope rules (for example, whether a type is private or not). Note also that the existence (i.e., the visibility) of the basic operations depends on the scope rules.

S4. 2.4 Real literals are literals of type *universal_real*. Integer literals are literals of type *universal_integer*.

Note that resolutions of real and integer literals include both those with and without potential implicit conversions.

S5. 3.2.1 The type of the initial expression in an object declaration must be the base type of the type mark.

S6. 3.2.2 The type of the static expression in a number declaration must be either *universal_integer* or *universal_real*.

S7. 3.5 If a range constraint is used in a subtype indication, either directly or as part of a floating point constraint or a fixed point constraint, the type of the expressions (or 'RANGE attribute bounds) must be the same as the base type of the type mark of the subtype indication.

S8. 3.5.4 If a range constraint is used as an integer type definition, each bound must have some integer type, but the two bounds need not have the same type.

S9. 3.5.5 For an attribute of the form T'POS(X), the operand X must have type T. For an attribute of the form T'VAL(X), the operand X may have any integer type, and the result type of the expression is T. For an attribute of the form T'VALUE(X), the operand X must have the predefined type STRING, and the result type is T. Attributes of the form T'SUCC(X) or T'PRED(X) require an operand of type T, and return a result of type T. For an attribute of the form T'IMAGE(X), the operand X must have type T, and the result is of the predefined type STRING.

S10. 3.5.7 The value in the expression after DIGITS, in a floating accuracy definition, must have an integer type. (The fact that the expression must be static is not used in overloading resolution.)

S11. 3.5.7 If a floating point constraint is used as a real type definition and includes a range constraint, then each bound of the range must be defined by an expression of some real type, but the two bounds need not have the same type.

- S12. 3.5.9 The value in the expression after DELTA in a fixed accuracy definition must belong to some real type.
- S13. 3.5.9 If a fixed point constraint is used as a real type definition, then each bound of the range must be defined by an expression of some real type, but the two bounds need not have the same type.
- S14. 3.6.1 RM 3.6.1/2 gives the following rules for overloading resolution of discrete ranges used in constrained array definitions, iteration rules, and declarations of entry families.

The type of the bounds must not be *universal_integer*.

If each bound is either a numeric literal, a named number, or an attribute, and the type of each bound is allowed to be both *universal_integer* and predefined INTEGER by other rules of the language, then the type of each bound is assumed to be INTEGER.

Otherwise, both bounds must have the same discrete type (other than *universal_integer*).

- S15. 3.6.1 In an index constraint in a subtype indication, the type of each discrete range must be that of the corresponding index of the base type of the type mark.
- S16. 3.6.2 The argument N used in the attribute designators for the attributes 'FIRST (N), 'LAST (N), 'RANGE (N), and 'LENGTH (N) must be an expression of type *universal_integer*.
- S17. 3.7 The default expression for a record component must have the type of the component.
- S18. 3.7.1 The default expression for a discriminant must have the type of the discriminant.
- S19. 3.7.2 Each expression in a discriminant constraint must have the type of the associated discriminant.
- S20. 3.7.3 Each choice in a variant in a record type definition must have the type of the discriminant.
- S21. 4.1 A name that is a simple name, a character literal, or an operator symbol must be directly visible (RM 8.3 and RM 8.4), except in a context clause (RM 10.1.1), in the parent unit name of a subunit (RM 10.2), or in a pragma argument association (RM 2.8 and RM 8.3).

The set of resolutions of such a name is the set of corresponding declarations that are directly visible at the occurrence of the name.

- S22. 4.1.1 An indexing operation is declared for each array type and access type whose designated type is an array type (RM 3.6.2/1 and RM 3.8.2/1). When the prefix of an indexed component is overloaded, the number and the types of the index expressions as well as the array component type can be used to help resolve the prefix. Similarly, the type of the prefix can resolve the type of the indexes. For example:

```

type A1 is array (1..2) of INTEGER;
type A2 is array (1..2, 1..2) of INTEGER;
type C1 is array (1..2) of CHARACTER;
function F return A1;
function F return A2;
function F return C1;

```



```

...
X : INTEGER := F(1);      -- unambiguous
Y : INTEGER := F(1, 2);   -- unambiguous
Z : CHARACTER := F(1);    -- unambiguous

```

Each index expression must have the corresponding index type. This requirement can be used to resolve overloadings within the index expression.

If the prefix denotes an entry family, the type of the index expression must be the type specified for the family.

Consider F.E(1) when the following declarations are visible:

```

task type T1 is
  entry E(1..2);
end T1;

task type T2 is
  entry E('A'..'B');
end T2;

function F return T1;
function F return T2;

```

Even though the declaration of an entry family cannot overload the entry family identifier, the prefix in F.E(1) can be overloaded, and the resolution of the prefix in this case is determined by the type of the index. In general, the resolutions also depend on the number and types of the actual parameters.

- S23. 4.1.2 The prefix of a slice must have as its type either a one-dimensional array type or an access type designating such a type. The bounds of the discrete range must have the type of the index.

The syntax and visibility rules determine whether a name is a slice or not, e.g., F(X..Y) is syntactically a slice, since X..Y can only be parsed as a range (not as an expression); similarly, F(X) is a slice if and only if X is a type mark, and this is determined by the visibility rules. For example:

```

type A1 is array (1..2) of INTEGER;
type A2 is array (1..2, 1..2) of INTEGER;
type A3 is array ('A'..'C') of INTEGER;
subtype SMALL is INTEGER range 1..1;

function F return A1;
function F return A2;
function F return A3;
function F (X : INTEGER) return INTEGER;
...
X1 : BOOLEAN := F(1..2) = F(1..2);      -- unambiguous
X2 : BOOLEAN := F(1) = F(2);             -- ambiguous
X3 : BOOLEAN := F(SMALL) = F(SMALL);     -- unambiguous

```

- S24. 4.1.3 The prefix of a selected component must either denote a package or an enclosing entity (rules RM 4.1.3 (e) and (f)), or it must have an access type (rule RM 4.1.3 (d)), or else it must be appropriate for a type with discriminants or a record type (rules RM 4.1.3 (a) and (b)) or a task type (rule RM 4.1.3 (c)). For rules (a), (b), (c), (e), and (f), the selector must be visible by selection after the dot.

Rule RM 4.1.3 (f) may only be applied inside the construct denoted by the prefix. In the case of an enclosing accept statement, the prefix must be the simple or the expanded name of a single entry or entry family (and not a renaming of the entry). If the prefix can denote more than one enclosing unit according to just the visibility rules, then the program is illegal regardless of the selector or the context.

If there is at least one possible interpretation of the prefix of a selected component as the name of an enclosing subprogram or an accept statement, then the prefix must not be a function call (regardless of the selector or context).

For rule (d), the prefix must be the name of an access value. For rules (a), (b), and (c), the prefix can be a name that denotes a value having a record, private, or task type, or a value of an access type that designates such a type. Note that, in general, prefixes can be either objects or function calls, or they can be names with prefixes that are objects, function calls, or names of values.

Note also that if any potential interpretation exists using rule RM 4.1.3 (f), that is the only possible interpretation (because (c) and (e) mean the same thing as (f), in case more than one applies).

Names declared by renaming declarations may be denoted using expanded names (AI-00187).

S25. 4.1.4 The declaration denoted by the prefix of an attribute must be determinable independently of the attribute designator and independently of the fact that it is the prefix of an attribute.

The rule above requires that the identity of the prefix be determinable independently of its context, and in particular, using only the fact that it is parsed as a prefix.

The following predefined attributes require a prefix that denotes either a type or a subtype. Therefore, the prefixes of these attributes (in a legal application) must be either type marks or expanded names. The visibility rules and the preference rule for expanded names (RM 4.1.3/19) fully determine the interpretation of the prefix for these attributes. The same would apply to implementation-defined attributes that always require the prefix to denote a type or subtype.

' AFT	' MACHINE_EMAX	' SAFE_EMAX
' BASE	' MACHINE_EMIN	' SAFE_LARGE
' DELTA	' MACHINE_MANTISSA	' SAFE_SMALL
' DIGITS	' MACHINE_OVERFLOW	' SMALL
' EMAX	' MACHINE_RADIX	' SUCC
' EPSILON	' MACHINE_ROUNDS	' VAL
' FORE	' MANTISSA	' VALUE
' IMAGE	' POS	' WIDTH
' LARGE	' PRED	

All of the other predefined attributes may (in certain contexts) have a prefix that is not an expanded name or a simple name and that contains an overloaded constituent. These are listed below:

' ADDRESS	' FIRST_BIT	' POSITION
' CALLABLE	' LAST	' RANGE
' CONSTRAINED	' LAST (N)	' RANGE (N)
' COUNT	' LAST_BIT	' SIZE
' FIRST	' LENGTH	' STORAGE_SIZE
' FIRST (N)	' LENGTH (N)	' TERMINATED

If the prefix is overloaded, resolution may not make use of the identity of the attribute, the

type of value or object required to serve as the prefix, the argument of the attribute (if any), or the fact that the prefix is the prefix of an attribute. In essence, the prefix of an attribute serves as a complete context for purposes of overloading resolution.

Of the second group of attributes, several are defined for prefixes that denote objects or components of objects:

```
'ADDRESS          'FIRST_BIT
'CONSTRAINED      'LAST_BIT
'SIZE             'POSITION
'SORAGE_SIZE
```

The prefixes of such attributes cannot be function calls, since function calls return values, not objects. Since this rule determines the syntactic structure of a program, it is applied independent of overloading resolution:

```
function F return INTEGER is
  X : SYSTEM.ADDRESS := F'ADDRESS;  -- unambiguous
```

F'ADDRESS is unambiguous because the parse of F as a function call is rejected. If there are two visible Fs, however, the prefix would be ambiguous:

```
function G (X : INTEGER) return INTEGER is
  function G return STRING;
  Y : SYSTEM.ADDRESS := G'ADDRESS;  -- ambiguous
```

If a function F returns an access value, then F.all denotes an object (see RM 4.1.3/11). If F is overloaded, every interpretation of F whose type is an access type must be considered. If there is more than one such interpretation, the prefix is illegal.

'FIRST_BIT, 'LAST_BIT, and 'POSITION require a prefix that denotes a component of a record object, rather than just an object. For these attributes, the selected component must be part of the prefix, so that the form may be F.Component or F.all.Component. With either form, the set of possible resolutions of F contains every access-to-record type in scope, and, in the case of F.Component, every record type, too (even though a component of a record value is not allowed as a prefix of these attributes.) For example:

```
type P_INT is access INTEGER;
type REC is
  record
    C1 : INTEGER;
  end record;
type P_REC is access REC;

function F return P_INT;      -- F1
function F return P_REC;     -- F2
function F return REC;       -- F3

A : INTEGER := F.all'SIZE;    -- ambiguous: F1, F2
B : INTEGER := INTEGER'(F.all'SIZE); -- ambiguous: F1, F2

X : INTEGER := F.C1'POSITION; -- ambiguous: F2, F3
Y : INTEGER := F.all.C1'POSITION; -- unambiguous
```

The declaration of A is ambiguous since there is more than one F that returns an access type. In the declaration of B, no more type information is present than is present in the declaration of A. X's declaration is ambiguous since there is no unambiguous use of F.C1 in any context (overloading resolution does not use information about whether a name denotes a value or a variable). The declaration of Y is legal, however, since only one F returns an access-to-record type.

Of the remaining attributes, 'CALLABLE and 'TERMINATED are defined for prefixes appropriate for a task type; 'FIRST(N), 'LAST(N), 'LENGTH, 'LENGTH(N), 'RANGE, and 'RANGE(N) are defined for prefixes appropriate for an array type or prefixes denoting a constrained array subtype; 'FIRST and 'LAST are defined for prefixes denoting scalar subtypes and constrained array subtypes, as well as prefixes appropriate for an array type. The following example illustrates some of the potential difficulties involved with these attributes.

```

type ARR  is array (1 .. 2) of INTEGER;
type ARR2 is array ('A' .. 'C') of INTEGER;
type ARR3 is array (1 .. 2, 1 .. 2) of INTEGER;
type P_ARR is access ARR;
type CHAR is new CHARACTER;

function F1 return ARR;
function F1 return P_ARR;

function F2 return ARR;
function F2 return CHAR;

function F3 return ARR;
function F3 return ARR2;
function F3 return ARR3;
function F3 (X : INTEGER) return ARR;

A : INTEGER := F1'FIRST;           -- ambiguous
B : INTEGER := F3(2)'FIRST;        -- ambiguous
C : CHARACTER := F3'FIRST;         -- ambiguous
D : INTEGER := F1.all'FIRST;       -- unambiguous

E : INTEGER := F2'FIRST;           -- ambiguous
F : INTEGER := INTEGER'(F2'FIRST); -- ambiguous
G : CHARACTER := F2'FIRST;         -- ambiguous
H : INTEGER := F3'FIRST(2);        -- ambiguous

```

Since the identity of the prefix must be resolved independently of the context, the requirement that the prefix of 'FIRST be appropriate for an array type may not be used for overloading resolution. The same holds true for the other array attributes. Similarly, for 'CALLABLE and 'TERMINATED (attributes that require a prefix appropriate for a task type), the requirement that the prefix be appropriate for a task type cannot be used for overloading resolution.

The prefix in A's declaration is ambiguous, since the parameter and the result type profile are insufficient to determine which F1 is being called. The prefix in B's declaration, too, cannot be resolved, since there is one F3 that can be called with a single parameter of type INTEGER, and another F3 whose result can be indexed with a single index of type INTEGER. Note that even though the result of subscripting F3's call yields a nonarray value, the prefix is considered ambiguous, since the appropriateness of the prefix for the attribute is not considered in resolving the prefix. F3'FIRST is ambiguous even though only one F3 can be called that will return an array whose first index is of type CHARACTER, i.e., the required result type for 'FIRST cannot be used to resolve the prefix. The prefix in D's declaration is unambiguous because there is only one F1 that returns an access type.

The expressions in the declarations of E, F, and G are ambiguous, even though only one F2 yields a result appropriate for an array type. The prefix in H's declaration is ambiguous, even though there is only one F3 that returns a two-dimensional array value.

S26. 4.2 The type of a string literal must be determinable solely from the context in which

the literal occurs, using only the fact that the string literal is a value of a one-dimensional array type whose component type is an enumeration type containing one or more character literals.

During the bottom-up phase, the set of resolutions of a string literal is the set of array types whose component type is a character type. The visibility of the character literals themselves is not used for overloading resolution. Even the identities of the characters in the string are irrelevant. This is significantly unlike the case for character literals, where the visibility rules are used to determine the set of potential resolutions. For example:

```

type A1 is array (1..2) of INTEGER;
B1 : BOOLEAN := "A" = "B";      -- unambiguous (1)

package P is
  type C1 is ('C');
end P;

type A2 is array (1..2, 1..2) of P.C1;
B2 : BOOLEAN := "C" = "C";      -- unambiguous (2)

type A3 is array (1..2) of P.C1;
B3 : BOOLEAN := "A" = "C";      -- ambiguous (3)

```

(1) is unambiguous because there is only one one-dimensional array type in scope whose component type is a character type, namely, the predefined type STRING. Hence, "A" and "B" are uniquely resolved to the type STRING. (2) is unambiguous because there is still only one one-dimensional array type in scope. (2) would be unambiguous even if we wrote "use P" before B2's declaration, thereby making C1's character literal directly visible. (3) is ambiguous because there are now two one-dimensional arrays in scope that have character component types and the equality operation for both types is directly visible. The fact that "A" and "C" are illegal string literals of type A3 is not relevant. (Note: "A" is an illegal string literal of type A3 since 'A' is not a value of type C1; "C" is illegal since 'C' is not directly visible.)

Note the difference between the treatment of string literals and array aggregates. If we replaced "A", "B", and "C" in the above example with their corresponding aggregates (e.g., if we wrote "A" as (1 => 'A')), then (1) is still unambiguous because there is only one nonlimited composite type in scope, namely, the type STRING. (2) is ambiguous, however, because there are now two composite types in scope and the equality operator is directly visible for both types. Hence, the context is not sufficient to determine the type of the aggregates.

- S27. 4.2 The type of the literal null must be determinable solely from the context in which the literal appears, using only the fact that the literal null is a value of an access type.

The set of potential resolutions for the literal null includes every access type.

- S28. 4.3 The type of an aggregate must be determinable solely from the context, using only the fact that the aggregate is a value of a nonlimited composite type.

The set of potential resolutions of an aggregate includes every nonlimited array or record type. The component simple names of a record aggregate must be visible by selection, but this fact is not used to resolve the aggregate, nor is the form of the choices. The names are used to resolve components of the aggregate during the top-down phase:

```

type REC1 is record
  X : INTEGER;
end record;

```

```

type REC2 is record
  Y : INTEGER;
end record;

type REC3 is record
  Z : FLOAT;
end record;

function F return INTEGER;      -- F1
function F return FLOAT;       -- F2
function F return REC1;        -- F3
...
(X => 1) = (X => 2)              -- ambiguous (1)
F = (X => F)                    -- unambiguous (2); F3 = (X => F1)

```

(1) is ambiguous because neither

- the component name used as the choice, nor
- the form of the name, nor
- the type of the expressions comprising the aggregate

can be used to determine the type of the aggregate. Hence, $(X \Rightarrow 1)$ can have type REC1, REC2, REC3, or STRING, and since equality operators are visible for each of these types, the expression cannot be resolved. The second expression is unambiguous because $(X \Rightarrow F)$ can only be a composite type, and there is only one visible F returning a composite type, namely, the type REC1. Hence, $(X \Rightarrow F)$ must have type REC1. Because the type of component X in REC1 is INTEGER, the call to F within the aggregate can also be resolved.

S29. 4.3.1 Each expression in a record aggregate must have the type of the associated record component(s).

Note that, conceptually, this rule is applied by the algorithm during the bottom-up phase. The sets of resolutions for each expression (and choice) of the aggregate are saved until the top-down phase, where one resolution is selected to resolve the component expressions. During the bottom-up phase, the set of resolutions of the aggregate must include all nonlimited composite types in scope, even those incompatible with the types of the component expressions.

S30. 4.3.1 In an array aggregate, each choice must have the corresponding index type, and each expression must have the component type.

Note that, conceptually, this rule is applied by the algorithm during the bottom-up phase. The sets of resolutions for each expression (and choice) of the aggregate are saved until the top-down phase, where one resolution is selected to resolve the component expressions. During the bottom-up phase, the set of resolutions of the aggregate must include all nonlimited types in scope, even those incompatible with the types of the component expressions.

S31. 4.4 The only names allowed as primaries are named numbers, attributes that return a value, and names denoting objects or values.

The only attributes that do not return a value are 'BASE, 'RANGE, and 'RANGE (N).

The type of an expression depends only on the type of its constituents and on the operators applied; for an overloaded constituent or operator, the determination of the constituent type, or the identification of the appropriate operator, depends on the context.

In an expression (or a simple expression), an operator must be directly visible (as for the corresponding operator symbol - see RM 8.3/18).

Names denoting subprograms and entries are not allowed as primaries. Function calls, though, are allowed. Primaries that are names of enumeration literals are parsed as parameterless function calls. Note that the resolution of a primary, factor, term, simple expression, relation, or expression is defined as its result type.

S32. 4.5.1 The short circuit control forms **and then** and **or else** are defined for two operands of the same boolean type and deliver a result of the same type. They are directly visible throughout their scope (RM 8.3/18).

S33. 4.5.2 The membership tests **in** and **not in** with a type mark are defined for all (nameable) types. The membership tests **in** and **not in** with a range are defined for all scalar types (including *universal_integer* and *universal_real*). The result type is predefined BOOLEAN, and the left operand must have the type of the type mark or the type of the bounds. Membership tests are directly visible throughout their scope (RM 8.3/18).

Note that for the membership test $X \text{ in } Y \text{ .. } Z$, the result type is BOOLEAN, and a unique resolution of the membership test operation is possible only if the resolution sets for X , Y , and Z have a single scalar type in common.

S34. 4.5.5 Predefined multiplication and division of operands of the same or different fixed point types deliver a result of type *universal_fixed*.

These operators are always directly visible. Note that extra parentheses are allowed (and ignored).

Type conversions such as $C(F * 1.1)$ are always ambiguous if F has a fixed type, because 1) *universal_real* is not a fixed point type, and hence, the 1.1 must be implicitly converted and 2) there are always at least two fixed types in scope (RM 3.5.9/7 and RM 9.6.3), so a unique implicit conversion cannot be found. Conversions such as $C(2 * 1.1)$, $C(2 * 2)$, and $C(2.0 * 1.1)$ are always ok, however, because nonuniversal resolutions are discarded (RM 4.6/15).

S35. 4.6 The type of an operand of an explicit type conversion must be determinable independently of the target type. This also applies to actual subprogram and entry parameters of mode **in out** or **out** that have the form of a type conversion.

S36. 4.6 An implicit conversion can only be (directly) applied if the operand is either a numeric literal, a named number, or an attribute returning a value of type *universal_integer* or *universal_real*. An implicit conversion is only applied if there is no interpretation of the context without such a conversion. The context must determine a unique result type for the conversion.

S37. 4.7 The operand of a qualified expression must have the type of the type mark. The result of a qualified expression has the base type of the type mark.

S38. 4.8 The type of the access value returned by an allocator must be determinable solely from the context, but using the fact that the value returned is of an access type having the named designated type.

The set of potential resolutions constructed during the bottom-up phase is the set of access types in scope designating the named type. Any constraints on the designated subtypes are ignored.

S39. 4.10 The predefined operations for the type *universal_integer* are the same as those for any integer type. The predefined operations for the type *universal_real* are

the same as those for any floating point type. In addition, the operators for multiplication of *universal_real* and *universal_integer* (in either order), and for division of *universal_real* by *universal_integer*, are included. The formal parameter names of the universal operators are LEFT and RIGHT (or just RIGHT) (RM 4.5).

- S40. 5.2 In an assignment statement, the named variable and the expression must have the same nonlimited type.
- Although the left side of an assignment operation must be a variable, this requirement cannot be used in overload resolution. For example, given F.I as the target of the assignment and assuming that F returns either a record value or an access value that designates a record, F.I is ambiguous if the type of component I is not sufficient to resolve the ambiguity.
- S41. 5.3 An expression specifying a condition in an if statement, while loop, exit statement, or selective wait must have a boolean type.
- S42. 5.4 The expression in a case statement must have a discrete type, but must otherwise be determinable independently of the context. Each choice must have the same type as the expression.
- S43. 5.8 The type of an expression in a return statement must be the base type of the type mark given in the function's specification.
- S44. 6.1 The default expression in a parameter specification must have the type of the corresponding formal parameter.
- S45. 6.4 The name in a procedure call must be the name of a procedure. The name in a function call must be the name of a function.
- S46. 6.4.1 In a subprogram call, each actual parameter must have the same type as the corresponding formal parameter.
- S47. 6.6 A call to a subprogram is illegal if the name of the subprogram, the number of parameter associations, the types and the order of actual parameters, the names of formal parameters, and the result type for functions, are not sufficient to identify exactly one subprogram or entry (see RM 9.5/5) declaration.

Note that the mode of a parameter is not relevant. Hence P(X+Y) would be ambiguous if the following declarations were visible:

```
procedure P (X : INTEGER);
procedure P (Y : in out INTEGER;
             Z : INTEGER := 0);
```

- S48. 8.5 In an object renaming declaration, the renamed object must have the type mark.
- S49. 8.5 In a subprogram or entry renaming declaration, the renamed subprogram or entry must have the same parameter and result type as the original subprogram specification.
- S50. 9.1 Within a task unit that declares a task type, a name declared in the task type must be used as a type mark.
- S51. 9.5 In an accept statement, the name of an entry, if any, must be the name of an indexed component; the type of the index expression must be the type of the discrete range in the entry family.

AD-A189 647

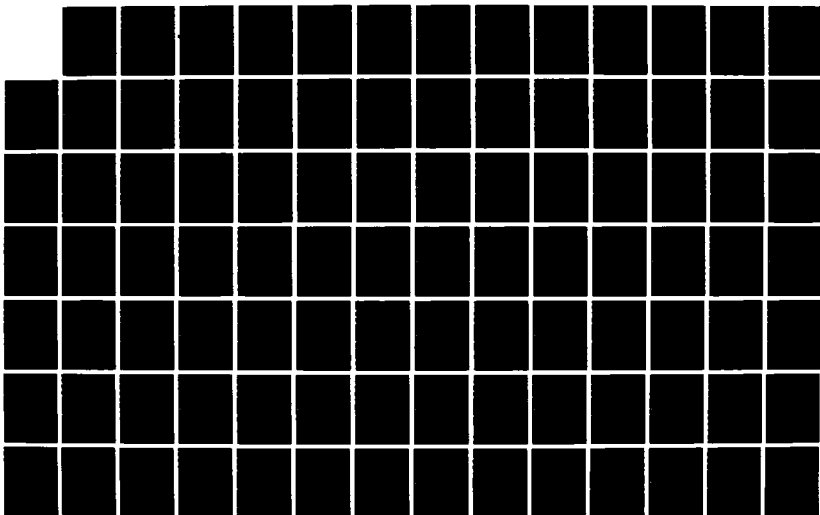
THE ADA (TRADE NAME) COMPILER VALIDATION CAPABILITY
IMPLEMENTERS' GUIDE VERSION 1(U) SOFTECH INC WALTHAM MA
J B GOODENOUGH DEC 86

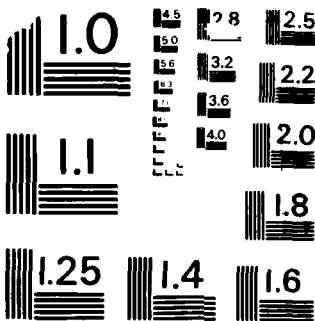
6/9

UNCLASSIFIED

F/G 12/5

NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

- S52. 9.5 A call to an entry is illegal if the name of the entry, the number of parameter associations, the types and the order of actual parameters, and the names of formal parameters, are not sufficient to identify exactly one procedure or entry declaration (when the entry call occurs in a conditional call or a timed wait, only entry declarations are considered).

The name in an entry call must be the name of an entry.

Overloading resolution is the same for procedures and entries.

- S53. 9.5 The declaration of the single entry or the entry family named by an accept statement must have the same parameter and result type profile as the formal part of the accept statement, and the simple name must be directly visible.
- S54. 9.6 The argument of the delay statement has the predefined fixed point type DURATION.
- S55. 9.10 The determination of the type of each task name in an abort statement uses the fact that the type of the name is a task type.
- S56. 12.1.1 The default expression for a generic formal in parameter must have the type of the parameter.
- S57. 12.3.1 A generic actual parameter of mode in or in out must have the type of the corresponding formal parameter.
- S58. 12.3.6 A generic actual subprogram parameter must be a subprogram, an enumeration literal, or an entry with the same parameter and result type profile as the corresponding formal.
- S59. 12.3.6 If a generic unit has a default subprogram parameter specified by a name, the name must denote a subprogram, an enumeration literal, or an entry that has the same parameter and result type profile as the formal parameter.
- S60. 12.3.6 If a generic unit has a default subprogram parameter specified by a box, there must be exactly one directly visible subprogram, enumeration literal, or entry with the same designator and parameter and result type profile at any instant at which that omits the corresponding actual parameter.
- S61. 13.2 In a length clause that specifies 'SIZE or 'STORAGE_SIZE, the expression must have some integer type.
In a length clause that specifies 'SMALL, the expression must have some integer type.
- S62. 13.3 The aggregate used in an enumeration representation clause must contain expressions of type *universal_integer*, and any choices must have the given enumeration type.
- S63. 13.4 Any expression contained in a record representation clause must have some integer type. The bounds of a range in a component clause need not have the same type.
The component name must be the name of a component of the record.
- S64. 13.5 The expression in an address clause must have the type SYSTEM.ADDRESS.
- S65. 13.5 An address clause is only legal if exactly one declaration with the simple name is directly visible.

Note that this rule makes overloaded occurrences of the simple name illegal. The set of

meanings is the single meaning of the unique, directly visible declaration denoted by the simple name.

Changes from July 1982

- S66. The RM now requires that overloading resolution provide a unique interpretation for every constituent of a complete context (not just a unique interpretation for a complete context).
- S67. A formal part is no longer listed as a complete context because the declarations comprising a formal part are already considered complete contexts.
- S68. The rules specifying how the expression in a case statement and the operand of a type conversion are treated are now given in RM 5.4/3, RM 4.6/3, and RM 8.7/12).
- S69. Syntax, scope, and visibility rules are to be used in resolving overloaded constructs.
- S70. Rules requiring a type to be nonlimited are to be used in overloading resolution.
- S71. Rules requiring the type of the prefix of an attribute to be determined independently of its context are now cited explicitly.
- S72. A reference to RM 8.4.1 ensures that when an actual in out or out parameter has the form of a type conversion, the variable's name must be resolved independently of the context.
- S73. The rules in RM 4.1.3/18, /19 are referenced without restricting the reference to enclosing functions.

Changes from July 1980

- S74. Numerous ambiguities regarding what contextual information was to be used in resolving overloaded constructs have now been cleared up.

Legality Rules

Overloading resolution must provide a unique interpretation for every constituent of a program unit.

Test Objectives and Design Guidelines

- T2. Check that because the type of the initialization expression in an object declaration must be the same as the object's base type (RM 8.7/8 and RM 3.2.1/1), an overloaded construct appearing in the initialization expression can be resolved.
- T3. Check that because the type of the expression in a number declaration must be either *universal_integer* or *universal_real* (RM 8.7/9 and RM 3.2.2/1), an overloaded construct in the expression can be resolved.
- T4. Check that because the expressions in a range used in a range constraint, fixed point constraint, or floating point constraint of a subtype indication must have the base type of the subtype indication's type mark (RM 8.7/8, RM 3.5/4, RM 3.5.7/14, and RM 3.5.9/13), an overloaded construct in the expressions can be resolved.
- T5. Check that because the bounds in the range of an integer type definition must have an integer type (RM 8.7/9 and RM 3.5.4/3), an overloaded construct appearing in the bounds can be resolved.

Check that for purposes of overloading resolution, an overloaded bound need not be static.

Check that for purposes of overloading resolution, the bounds need not have the same integer type.

- T6. Check that because there exists an implicit conversion that converts a *universal_integer* value into the corresponding value of an integer type (RM 8.7/13 and RM 3.5.5/1), an overloaded call can be resolved (see also T37).
- T7. Check that an overloaded construct can be resolved because:
- for an attribute of the form T'POS(X), the operand X must have type T, and the result is of type *universal_integer* (RM 8.7/8 and RM 3.5.5/6).
 - for an attribute of the form T'VAL(X), the operand X may have any integer type, and the result is of type T (RM 8.7/8 and RM 3.5.5/7).
 - for an attribute of the form T'VALUE(X), the operand X must have the predefined type STRING, and the result is of type T (RM 8.7/8 and RM 3.5.5/12).
 - for attributes of the form T'SUCC(X) and T'PRED(X), the operand X must have type T and the result is of type T (RM 8.7/8 and RM 3.5.5/8, 9).
 - for an attribute of form T'IMAGE(X), the operand X must have type T, and the result is of the predefined type STRING (RM 8.7/8 and RM 3.5.5/10).
- T8. Check that because there exists an implicit conversion that converts a convertible operand of type *universal_real* into a value of a real type (RM 8.7/13 and RM 3.5.6/5), an overloaded call can be resolved.
- T9. Check that an overloaded construct can be resolved because:
- in a floating point type definition or subtype indication, the digits expression must have some integer type (RM 8.7/9 and RM 3.5.7/3); check that for purposes of overloading resolution, it is irrelevant whether the overloaded construct is static or not.
 - in a fixed point type definition or subtype indication, the delta expression must have some real type (RM 8.7/9 and RM 3.5.9/3); check that for purposes of overloading resolution, it is irrelevant whether the overloaded construct is static or not.
- T10. Check that because both bounds of a range constraint of a floating point type definition must have some real type (RM 8.7/9 and RM 3.5.7/3), an overloaded construct appearing in a bound can be resolved.
- Check that for purposes of overloading resolution, it is irrelevant whether an overloaded bound is static or not.
- Check that for purposes of overloading resolution, both bounds need not have the same real type.
- Check that the bounds of the range may not have type *universal_fixed*.
- T11. Check that because the delta expression in a fixed point type definition must have some real type (RM 8.7/9 and RM 3.5.9/3), an overloaded construct in the expression can be resolved; check that for purposes of overloading resolution, it is irrelevant whether the overloaded construct is static or not.
- Check that the delta expression may not have type *universal_fixed*.
- T12. Check that because both bounds in a range constraint of a fixed point type definition must have some real type (RM 8.7/9 and RM 3.5.9/3), an overloaded construct appearing in one of the bounds can be resolved.

Check that for purposes of overloading resolution, it is irrelevant whether a bound is static or not.

Check that for purposes of overloading resolution, both bounds do not have to have the same real type.

Check that the bounds of the range may not have type *universal_fixed*.

T13. Check that an overloaded construct contained in the bounds of a discrete range used in a constrained array type definition, an iteration rule, or an entry family declaration can be resolved because:

- the bounds of the discrete range must have the same type and the type must be discrete (RM 8.7/8, RM 8.7/9, and RM 3.6.1/2).

Implementation Guideline: Include the use of overloaded enumeration literals.

- if both bounds of the discrete range are either integer literals, named numbers, or attributes, both are *potentially* of type *universal_integer*, and there is more than one integer type whose scope includes the discrete range, then an implicit conversion to predefined INTEGER type is assumed for both bounds (RM 8.7/8 and RM 3.6.1/2).

Check that when the operations for more than one integer type are visible, the implicit conversion of bounds to the predefined INTEGER is not permitted if the bounds are not integer literals, named numbers, or attributes returning nonuniversal values (e.g. 'SUCC). In particular, check that static expressions using predefined operators are not allowed.

T14. Check that because the type of each discrete range of an index constraint used in a subtype indication must be the same as that of the corresponding index of the base type of the type mark (RM 8.7/8 and RM 3.6.1/3), an overloaded expression in the index constraint can be resolved.

Check that *universal_integer* literals, named numbers, and attributes are converted to the index base type.

T15. Check that because the argument N used in the attributes 'FIRST(N), 'LAST(N), 'RANGE(N), and 'LENGTH(N) must have type *universal_integer* (RM 8.7/9 and RM 3.6.2/2), an overloaded construct appearing in the argument can be resolved.

T16. Check that because the default expression for a record component must have the type of the component (RM 8.7/8 and RM 3.7/5), an overloaded construct used in the default expression can be resolved.

T17. Check that because the default expression for a discriminant must have the type of the discriminant (RM 8.7/8 and RM 3.7.1/4), an overloaded construct appearing in the default expression can be resolved.

T18. Check that because each expression in a discriminant constraint must have the type of the associated discriminant (RM 8.7/8 and RM 3.7.2/4), an overloaded construct appearing in the expression can be resolved.

T19. Check that because the simple expressions and range bounds of variant choices in record type definitions must have the type of the discriminant used for the variant part (RM 8.7/8 and RM 3.7.3/3), an overloaded construct appearing in a choice can be resolved.

Check that for purposes of overloading resolution, it is irrelevant whether the overloaded construct is static or not.

T20. Check that overloading resolution does not use the fact that the 'CONSTRAINED attribute

is defined only for objects with discriminants and private types. The identity of the prefix must be determinable independently of the context.

T23. Check that an overloaded prefix or index expression can be resolved because:

- the prefix of an indexed component must either denote an entry family, or its type must be an array type or an access type whose designated type is an array type (RM 8.7/9 and RM 4.1.1/3).
- for entry families, there must be a single expression having the type of the family index (RM 8.7/9 and RM 4.1.1/3).
- for array types, there must be as many expressions as array dimensions, and each expression must have the corresponding index type (RM 8.7/8 and RM 4.1.1/3).

Implementation Guideline: In at least one case, both the prefix and index types should be ambiguous when considered independently.

- the required type of the indexed component.

T24. Check that an overloaded prefix or bound of a discrete range can be resolved because:

- the type of the prefix of a slice must be a one-dimensional array type or an access type whose designated type is a one-dimensional array type (RM 8.7/10 and RM 4.1.2/3).
- the bounds of the discrete range must have the index type (RM 8.7/8 and RM 4.1.2/3).
- the required type of the slice value.

T25. Check that an overloaded prefix of a name that is a selected component can be resolved because:

- the type of the prefix has a discriminant or is an access type whose designated type has a discriminant, and the selector is the name of the discriminant (RM 8.7/10 and RM 4.1.3/5).
- the type of the prefix is a record type or an access type whose designated type is a record type, and the selector is the name of a component of the type (RM 8.7/10 and RM 4.1.3/7).
- the type of the prefix is a task, a task type, or an access type whose designated type is a task type, and the selector is the name of a single entry or an entry family of the task (RM 8.7/10 and RM 4.1.3/10).
- the type of the prefix is an access type and the selector is the reserved word *all* (RM 8.7/8 and RM 4.1.3/12).
- the prefix of an expanded name cannot denote a name declared by a renaming declaration (RM 8.7/13 and RM 4.1.3/18).
- if the prefix can be interpreted as the name of an enclosing unit, then all other possible interpretations are rejected (RM 8.7/13 and RM 4.1.3/19).

Check that a prefix of a selected component cannot be resolved based on the selector if the prefix potentially denotes two or more enclosing entities (RM 4.1.3/18).

Implementation Guideline: Check for enclosing subprograms and accept statements.

T26. Check that the meaning of the prefix of an attribute must be determinable independently of

the attribute designator and independently of the fact that it is the prefix of an attribute. In particular, check that the following legality constraints are not used for overloading resolution:

- Check that for 'SIZE, 'FIRST_BIT, 'LAST_BIT, and 'POSITION, the requirement that the prefix denote an object rather than a value is not used for overloading resolution.
- Check that for 'CALLABLE and 'TERMINATED, the requirement that the prefix be appropriate for a task type is not used for overloading resolution.
- Check that for 'FIRST, 'FIRST(N), 'LAST, 'LAST(N), 'LENGTH, 'LENGTH(N), 'RANGE, and 'RANGE(N), the requirement that the prefix be appropriate for an array type is not used for overloading resolution.
- Check that even though the prefix of the attribute 'COUNT must denote an entry of a task, this information is not used to resolve any overloaded constituents of the prefix (RM 8.7/12 and RM 4.1.4/3).
- Check that 'ADDRESS can have a parameterless function as its prefix, and this use is considered unambiguous if the function is not overloaded.

T27. Check that because a string literal is a value of a one-dimensional array type whose component type is an enumeration type containing at least one character literal (RM 8.7/11 and RM 4.2/4), an overloaded call can be resolved.

Implementation Guideline: Use a case where the corresponding array aggregate would not be resolvable.

Check that the character literals comprising the string literal are not used to resolve the type of the literal (RM 8.7/12 and RM 4.2/4).

T28. Check that because the type of the literal null must be an access type (RM 8.7/11 and RM 4.2/4), an overloaded call can be resolved.

Implementation Guideline: Use a case in which there is only one access type declared.

Check that if more than one access type is declared, the literal null is ambiguous and its type must be determined from the context in which it appears (RM 8.7/12 and RM 4.2/4).

T29. Check that because an aggregate must be a composite, nonlimited type (RM 8.7/12 and RM 4.3/7), an overloaded call can be resolved.

Implementation Guideline: Check that the form of choices in an aggregate and the type of the choice or the expression is not used to resolve the type of the aggregate.

Check that because single-component aggregates must use named associations, an overloaded call can be resolved (RM 8.7/7 and RM 4.3/4).

Check that because a private type with discriminants is not a composite type (RM 3.3/2), an overloaded call with an aggregate can be resolved.

T30. Check that because the expression of a record component association must have the type of the associated record components (RM 8.7/8 and RM 4.3.1/1), an overloaded expression in a record aggregate can be resolved.

T31. Check that because the type of each choice in an array aggregate must be the same as the type of the corresponding index (RM 8.7/8 and RM 4.3.2/1), an overloaded expression used as a choice in an array aggregate can be resolved.

Check that an overloaded expression used as a value in an array component association must have the type of the array component (RM 8.7/8 and RM 4.3.2/1).

T32. Check that an overloaded expression used as the parameter of certain attributes can be resolved because:

- for attributes of the form T'SUCC(X), T'PRED(X), T'POS(X), and T'IMAGE(X), the expression must have type T (RM 8.7/8 and RM 3.5.5/8, /9, /6, and /10).
- for an attribute of the form T'VAL(X), the expression must have an integer type (RM 8.7/8 and RM 3.5.5/7).
- for an attribute of the form T'VALUE(X), the expression must have the predefined type STRING (RM 8.7/8 and RM 3.5.5/12).

Check that an overloaded construct can be resolved because:

- attributes of the form T'SUCC(X), T'PRED(X), T'VAL(X), and T'VALUE(X) deliver a result of type T (RM 8.7/11 and RM 3.5.5/8, /9, /7, and /12).
- the attribute T'POS delivers a result of type *universal_integer* (RM 8.7/11 and RM 3.5.5/6) (which can be implicitly converted to any integer type; RM 4.6/15).
- the attribute T'IMAGE delivers a result of type STRING (RM 8.7/11 and RM 3.5.5/10).

T33. Check that an overloaded construct can be resolved because the short circuit control forms and then and or else

- are defined only for two operands of the same boolean type (RM 8.7/8 and RM 4.5.1/4);
- deliver a result of the same type as the operands (RM 8.7/11 and RM 4.5.1/4).

T34. Check that an overloaded construct can be resolved because:

- the result type of the membership tests In and not In is predefined BOOLEAN (RM 8.7/11 and RM 4.5.2/10).
- in a membership test with a range, both bounds of the range and the left operand of the test must have the same type (RM 8.7/8 and RM 4.5.2/10), and this must be a scalar type (RM 8.7/9 and RM 4.5.2/10).
- In a membership test with a type mark, the type of the left operand must be the base type of the type mark (RM 8.7/8 and RM 4.5.2/10).

T35. Check that an overloaded construct can be resolved because if one operand of a predefined multiplication operator has a fixed point type, the other operand:

- must have the predefined type INTEGER if the result of the multiplication is not explicitly converted to some numeric type (RM 8.7/8 and RM 4.5.5/7);
- must have a fixed point type or the predefined type INTEGER if the result of the multiplication is explicitly converted to some numeric type (RM 8.7/8, RM 4.5.5/7, and RM 4.5.5/10, /11).

Check that an overloaded construct can be resolved because, if the first operand of a predefined division operator has a fixed point type, the second operand:

- must have the predefined type INTEGER if the result of the division is not explicitly converted to some numeric type (RM 8.7/8 and RM 4.5.5/7);
- must have a fixed point type or the predefined type INTEGER if the result of the division is explicitly converted to some numeric type (RM 8.7/8, RM 4.5.5/7, and RM 4.5.5/10, /11).

Check that an overloaded expression used as the exponent operand of an integer or a floating point exponentiation must have the type predefined INTEGER (RM 8.7/8 and RM 4.5.6/5).

Check that because there are always two fixed point types in scope, fixed point multiplication or division cannot have an operand of type *universal_real* (since there is no unique implicit conversion to a fixed point type) (RM 4.5.6/5 and RM 4.6/15) (see IG 4.5.5.b/T38).

T36. Check that the type of an operand of an explicit type conversion must be determinable independently of the target type. This also applies to actual subprogram and entry parameters of mode in out or out that are type conversions (RM 8.7/12, RM 4.6/3, and RM 6.4.1/3).

T37. Check that an overloaded construct can be resolved because an implicit conversion of a value of the type *universal_integer* to another integer type, or of a value of type *universal_real* to another real type, can only be applied if the operand is either a numeric literal, a named number, or an attribute, and if and only if the innermost complete context determines a unique (numeric) target type for the implicit conversion, and there is no legal interpretation of this context without this conversion (RM 8.7/13 and RM 4.6/15).

T38. Check that because the operand of a qualified expression must have the type of the type mark (RM 8.7/8 and RM 4.7/3), an overloaded construct appearing in the operand can be resolved.

T39. Check that because an allocator returns an access type whose designated type has the same base type as the base type named in the allocator (RM 8.7/11 and RM 4.8/3), an overloaded call can be resolved.

Check that if the name of the designated type is not sufficient to resolve the type of the allocator, the type must be determined by the context (RM 8.7/11 and RM 4.8/3).

T40. Check that because the same operations are predefined for the type *universal_integer* as for any integer type (RM 8.7/8 and RM 4.10/2), an overloaded construct can be resolved.

Check that because the same operations are predefined for the type *universal_real* as for any floating point type (RM 8.7/8 and RM 4.10/2), an overloaded construct can be resolved.

Check that because the operations predefined for *universal_integer* and *universal_real* include multiplication of *universal_integer* and *universal_real* (in either order), and division of *universal_real* by *universal_integer* (RM 8.7/8 and RM 4.10/2), an overloaded construct can be resolved.

T41. Check that because the named variable and the expression in an assignment statement must have the same nonlimited type (RM 8.7/8 and RM 5.2/1), an overloaded construct appearing in the variable or the expression can be resolved.

T42. Check that because an expression specifying a condition in an if statement, while loop, exit statement, or selective wait must have a boolean type (RM 8.7/8 and RM 5.3/3), an overloaded construct appearing in the expression can be resolved.

T43. Check that because the expression in a case statement must have a discrete type (RM 8.7/9 and RM 5.4/3), an overloaded construct appearing in the expression can be resolved.

Check that the type of the choices cannot be used to resolve the type of the case expression (RM 8.7/12 and RM 5.4/3).

Check that the fact that the choices in a case statement must be static cannot be used to resolve the type of the choices or the type of the case expression (RM 8.7/7).

Check that because each choice must have the same type as the expression (RM 8.7/8 and RM 5.4/3), an overloaded construct in a choice of a case statement can be resolved.

- T44. Check that because the type of an expression in a return statement must be the base type of the type mark given in the specification of the function (RM 8.7/8 and RM 5.8/5), an overloaded construct appearing in the expression of a return statement can be resolved.
- T45. Check that because the default expression in a parameter specification of a subprogram or an entry must have the type of the corresponding formal parameter (RM 8.7/8 and RM 6.1/4), an overloaded construct appearing in the expression can be resolved.
- T46. Check that the name in a procedure call, entry call, or function call can be resolved because (RM 8.7/7):
- the name in a procedure call must be the name of a procedure (rather than the name of a function);
 - the name in an entry call must be the name of an entry (rather than the name of a function);
 - the name in an entry call appearing in a conditional or timed entry call must be the name of an entry (rather than the name of a procedure or function); and
 - the name in a function call must be the name of a function (rather than the name of a procedure or entry).

Check that the use of a named actual parameter in a function call allows a function call to be distinguished from an indexed component (RM 8.7/7).

Implementation Guideline: Check that when $F(X)$ is either an indexed component or a function call, $F(\text{PARAM} \Rightarrow X)$ is unambiguously a function call.

- T47. Check that because each actual parameter in a subprogram call must have the same type as the corresponding formal parameter (RM 8.7/8 and RM 6.4.1/1), an overloaded actual parameter or subprogram name can be resolved.

Implementation Guideline: Note that the combination of possible actual parameter types might serve to resolve the name of the subprogram, and hence, the type of each actual parameter.

- T48. Check that a call to a subprogram is illegal if the name of the subprogram, the number of parameter associations, the types and the order of actual parameters, the names of formal parameters, and the result type for functions are not sufficient to identify exactly one subprogram declaration (RM 8.7/13 and RM 6.6/3).

Check that the order of named associations in a subprogram is not significant to the process of overloading resolution.

Check that the mode of a parameter is not used to resolve an overloaded call of a subprogram or an entry.

- T49. Check that because the renamed object in an object renaming declaration must have the type of the type mark (RM 8.7/8 and RM 8.5/4), an overloaded prefix in the name for the renamed object can be resolved.

- T50. Check that because a renamed subprogram or entry must have the same parameter and result type profile as the given subprogram specification (RM 8.7/8, RM 8.7/19, and RM 8.5/7), an overloaded subprogram or entry name can be resolved.

Implementation Guideline: Check this is so even when the renamed subprogram is an attribute or an enumeration literal.

Check that attributes that have a parameter or result of a universal type cannot be renamed in a renaming declaration.

Check that multiplication and division operators for fixed point types cannot be renamed in a renaming declaration.

- T51. Check that because the type of the index expression in an accept statement for an entry of a family must be the same as that of the discrete range in the entry family declaration (RM 8.7/8 and RM 9.5/4), an overloaded construct appearing in the index expression can be resolved.

- T52. Check that an entry call is illegal if the name of the entry, the number of parameter associations, the types and the order of the actual parameters, the names of the formal parameters, and the result type (if any) are not sufficient to identify exactly one entry declaration (RM 8.7/8, RM 8.7/19, and RM 6.6/3).

Check that the occurrence of an entry call in a conditional or timed entry call statement determines that an entry is being called rather than a procedure (see IG 8.7.b/T46).

- T53. Check that because the single entry named by an accept statement must have the same parameter and result type profile as the formal part of the accept statement (RM 8.7/8 and RM 9.5/7), it can be determined which entry is denoted by the entry name in an accept statement.

Implementation Guideline: Note that since entry family names cannot be overloaded, there is no potential ambiguity in deciding which entry family is denoted by an accept statement for an entry family member.

- T54. Check that because the argument of a delay statement must have the predefined fixed point type DURATION (RM 8.7/8 and RM 9.6/3), an overloaded construct appearing in the delay expression can be resolved.

- T56. Check that because each name in an abort statement must denote an object of a task type (RM 8.7/9 and RM 9.10/3), an overloaded constituent in a name can be resolved.

- T57. Check that because the default expression for a generic formal in parameter must have the type of the parameter (RM 8.7/8 and RM 12.1.1/2), an overloaded construct appearing in the default expression can be resolved.

- T58. Check that because a generic actual parameter of mode in or in out must have the type of the corresponding formal parameter (RM 8.7/8, RM 12.3.1/1, and 12.3.1/2), an overloaded construct appearing in the actual parameter can be resolved.

Implementation Guideline: In particular, check string literals that are identical to operator symbols.

- T59. Check that because a generic actual subprogram parameter must be a subprogram, an enumeration literal, or an entry with the same parameter and result type profile as the corresponding formal parameter (RM 8.7/8, RM 8.7/19, and RM 12.3.1/1), an overloaded name appearing as an actual parameter can be resolved.

Implementation Guideline: In particular, check string literals that are identical to operator symbols.

- T60. If a generic unit has a default subprogram parameter specified by a name, check that because this name must denote a subprogram, an enumeration literal, or an entry with the same parameter and result type profile as the formal parameter (RM 8.7/8, RM 8.7/19, and RM 12.3.6/2), an overloaded default name can be resolved.

- T61. If a generic unit has a default subprogram parameter specified by a box, check that because there must be exactly one directly visible subprogram, enumeration literal, or entry with the same designator the same parameter and result type profile as the formal subprogram at any instantiation that omits the corresponding actual parameter (RM 8.7/8, RM 8.7/19, and RM 12.3.6/3), the appropriate default name can be found when more than one name is visible.

Implementation Guideline: Include a check for names made visible by a use clause.

T62. Check that an overloaded construct can be resolved because:

- in a length clause that specifies 'SIZE or 'STORAGE_SIZE, the expression must have some integer type (RM 8.7/9, RM 13.2/5, and RM 13.2/8).
- in a length clause that specifies 'SMALL, the expression must have some real type (RM 8.7/9 and RM 13.2/12).

T63. Check that because the aggregate used in an enumeration representation clause must contain expressions of type *universal_integer*, and any choices must have the given enumeration type (RM 8.7/8 and RM 13.3/3), an overloaded construct appearing in the aggregate can be resolved.

T64. Check that because any expression contained in a record representation clause must have some integer type (RM 8.7/9 and RM 13.4/3), an overloaded construct appearing in such expressions can be resolved.

Check that the bounds of a range in a component clause need not have the same type (RM 13.4/3).

T65. Check that because the expression in an address clause must have type *SYSTEM.ADDRESS* (RM 8.7/8 and RM 13.5/3), an overloaded construct appearing in an address clause can be resolved.

Chapter 9

Tasks

9.1 Task Specifications and Task Bodies

Semantic Ramifications

S1. Within the body of task type T, the identifier T cannot be used as a type mark denoting the task type, i.e., T cannot be used as the type mark in a conversion, a qualification, a membership operation, a generic actual type parameter, a declaration of a formal parameter of a subprogram or generic unit, a renaming declaration, an access type declaration, an allocator, an array or record component definition, a derived type definition, an object declaration, or a subtype declaration. Although the use of the type mark is forbidden in these contexts, a different identifier denoting the task type is allowed, e.g.:

```
task type T;
type ACC_T is access T;

subtype ST is T;           -- new identifier denoting T

task body T is
  X : T;                   -- illegal
  Y : ST;                   -- legal, but recursive
  Z : ACC_T;
begin
  if ... then
    Z := new T;             -- illegal
    Z := new ST;            -- legal
```

S2. Although a task unit's name cannot be used as a type mark within its own body, it can be used as the prefix of an expanded name, e.g., T.X is a legal name within T's body in the previous example. The task unit's name can also be used as the name of a task object in an abort statement, as an actual in or in out parameter to a subprogram or a generic unit, as the prefix to the 'CALLABLE and 'TERMINATED attributes, and as the expression value in a membership test.

S3. If a subtype declaration declares a new name for a task type, the new name denotes a type, not a task unit (RM 3.3.2/1). For example, in the above program segment, ST.X would not be a legal expanded name within (or outside) the body of T.

S4. The only representation clause allowed in a task specification is an address clause for an entry. The only other representation clauses that apply to tasks or to task types are the 'STORAGE_SIZE and 'SIZE length clauses, but these cannot be used inside a task specification, since they and the entity to which they apply must both occur immediately within the same declarative part (or package specification) (RM 13.1/5).

S5. Neither a task declaration nor a task body can be a library unit (RM 10.1/2). A task body, however, can be a subunit (see RM 10.2/2) and thus can be a compilation unit.

S6. Only the following predefined pragmas may appear in a task specification: LIST, PAGE, and PRIORITY.

S7. The main program is considered a subprogram called by some environment task. This rule allows the main program to suspend its own execution by executing a delay statement.

Changes from July 1982

S8. Use of the name of a task unit as a type mark is not allowed within the task unit itself.

Changes from July 1980

S9. There are no significant changes.

Legality Rules

- L1. A task body must be declared for every task or task type declared by a task specification (RM 9.1/1).
- L2. A task body cannot be declared unless the corresponding task specification was given previously in the same declarative part (RM 3.9/9), or, if the task body is given in the declarative part of a package body, the corresponding task specification was given as a declarative item in a corresponding package specification (RM 7.1/4) or earlier in the same declarative part (RM 3.9/9).
- L3. If an identifier is given at the end of a task body or specification, it must be the same as the identifier for the task unit (RM 9.1/4).
- L4. Within a task unit, the name of the task cannot be used as a type mark (RM 9.1/4).
- L5. An address clause is the only form of representation clause allowed in a task specification (RM 13.5/7). Such a clause must name an entry declared earlier in the task specification (RM 13.5/7).

Exception Conditions

- E1. `CONSTRAINT_ERROR` can be raised when the discrete range of an entry declaration is elaborated. Specifically, for a discrete non-null range of the form `ST range L .. R`, `CONSTRAINT_ERROR` is raised if L or R is outside the range of ST but within the range of ST's base type (see RM 3.6.1/4, RM 3.5/4, and RM 3.3.2/9).
- E2. If an entry declaration has a discrete range of the form `L .. R`, where L and R are integer literals, named numbers having type *universal_integer*, or attributes returning a value of type *universal_integer*, `NUMERIC_ERROR` is raised if either L or R lies outside the range of `INTEGER` (RM 4.6/15 and RM 3.6.1/2).

Test Objectives and Design Guidelines

- T1. Check that
 - a parameter list may not follow the identifier in a task specification.
 - a task specification cannot be a compilation unit.
 - the identifier at the end of a task specification or body cannot be different from the identifier serving as the name of the task.
 - two entry declarations cannot be separated by an address clause that applies to the first entry declaration.
 - a task specification is allowed to have no entry declarations (2 ways).
 - a length clause is not allowed in a task specification (for 'SIZE, 'STORAGE-SIZE, or 'SMALL).
- T2. Check that inappropriate declarations are not allowed in a task specification, namely, a declaration of a variable, constant, access type, subtype, array type, record type, procedure, function, package, task, or exception is not allowed.

Check that inappropriate pragmas are ignored, namely, CONTROLLED, INLINE, INTERFACE, MEMORY_SIZE, OPTIMIZE, PACK, and STORAGE_UNIT.

- T3. Check that if a task specification is given in a package specification, a corresponding task body must be provided in the package body.

Implementation Guideline: Try providing a body in a nested package body as well as simply omitting a body.

Check that if a task specification is given in a declarative part, a corresponding body must be provided in the same declarative part.

Check that a task body cannot be provided in a declarative part of a package or subprogram if there has been no preceding task specification.

Implementation Guideline: Include a check that the task specification cannot follow the task body.

- T4. Check that the name of a task type cannot be used within its own body in a conversion, a qualification, a membership operation, a generic actual parameter (corresponding to a formal type parameter), a declaration of a formal parameter of a subprogram or generic unit, a renaming declaration, an access type declaration, an allocator, an array or record component definition, a derived type definition, an object declaration, or a subtype declaration.

Implementation Guideline: For the generic actual parameter case, use a subtype name to declare the corresponding formal parameter.

Implementation Guideline: Include some checks in a nested task body.

Check that within a task type's body, the name of the enclosing task unit refers to the object whose designated task is executing the body, and that a different identifier denoting the task type cannot be used as the name of task object.

Implementation Guideline: Declare a new identifier for a task type by using a subtype declaration. (A new identifier cannot be created for a unit that declares a single task.)

Implementation Guideline: Check the use of a task's name within its own body in an abort statement, a membership test, and in the attributes 'CALLABLE and 'TERMINATED.

Check that a subtype name denoting a task type can be used inside its own body as a type mark, but not as the prefix in an expanded name.

- T5. Check that the identifier of a single task cannot be used as a type mark.
- T6. Check that entry declarations are elaborated when the task specification is elaborated (not when the task is activated), and are elaborated in the order given in the source code.
- T7. Check that if the elaboration of an entry declaration raises CONSTRAINT_ERROR, no tasks are activated and TASKING_ERROR is not raised.

9.2 Task Types and Task Objects

Semantic Ramifications

S1. For most purposes, it is convenient to think of a task object as holding a pointer whose value gives access to the code of a task. It is partly for this reason that a task object is said to designate a task.

S2. Access types can be declared whose designated types are tasks:

```
task type T;
type ACC_T is access T;
X : ACC_T;
```

The object designated by X is a task object, which in turn designates a task. X.all denotes this task object.

S3. Functions can return task objects as values, e.g.:

```
ARR : array (1..10) of TASKS;
function SELECT (FROM : INTEGER) return TASKS is
begin
    return ARR (FROM);
end SELECT;
```

The function can be used in an entry call, e.g., SELECT(5).E;.

S4. Task objects may be components of other objects, and may be objects (or components of objects) designated by access types. Task objects and objects having subcomponents of a task type may be actual in or in out parameters of subprograms and entries. There is no semantic difference between passing a task as an in or as an in out parameter. In both cases, the task serving as the actual parameter is accessible via the formal parameter. In particular, such a task is not reactivated:

```
task type T is
    entry STATE1;
    entry STATE2;
end T;

OBJ : T;

task body T is
begin
    accept STATE1;
    accept STATE2;
end T;

procedure P (X : T) is
begin
    X.STATE2;      -- accepted if task X not just activated
end P;

...
OBJ.STATE1;
P (OBJ);
```

Because the formal and the actual parameter designate the same task, X.STATE2 will be accepted when P is called. The same effect occurs for mode in out.

S5. The mode out is disallowed for task types and for types having a subcomponent of a task type since RM 7.4.4/4 says, in effect, that if an out parameter is limited, it must be a limited private type whose full declaration is not limited. In short, because of RM 7.4.4/4, it is not possible to create a subprogram or an entry with an out parameter that is either a task type, a composite type with a subcomponent having a task type, or a limited type whose full declaration is such a type.

S6. Although assignment is not defined for task types, a function yielding a task value or a task object can be given as the default initial expression of a formal subprogram parameter, since the := in such a declaration does not represent assignment, but merely the correspondence between a formal and actual parameter (RM 7.4.4/10).

S7. The full declaration of a limited private type can be a task type declaration (RM 7.4.1/3). In such a case, the task entries (if any) are not visible outside the package.

S8. Since predefined equality and assignment are not defined for a task type, a task type is a

limited type (RM 7.4.4/1), and so is any record or array type having a subcomponent of a task type (RM 7.4.4/2).

S9. If a type is derived from a task type, the derived type is a task type and is subject to the usual rules for task types (RM 3.4/4 and RM 3.4/7).

S10. The amount of data storage made available to a task is initially determined when the object is created, i.e., when an object declaration is elaborated or when an allocator is evaluated. In the case of an object declaration, the amount of storage is determined before the task is activated (although it might change after task activation if the amount of storage for a task is determined dynamically):

```

OBJ : TSK_TYPE;
package PACK is
    STORE : INTEGER := OBJ' STORAGE_SIZE;    -- OBJ not yet activated
end PACK;

```

Changes from July 1982

S11. An out parameter of a procedure or an entry cannot have a task type or a composite type with a subcomponent of a task type.

Changes from July 1980

S12. Task objects are not allowed as generic in parameters.

S13. An in out parameter of a subprogram or an entry can have a task type.

Legality Rules

L1. Neither a task type nor a composite type with a subcomponent having a task type is allowed to be the type of an out formal parameter (RM 9.2/1 and RM 7.4.4/4).

L2. Neither a task type nor a composite type with a subcomponent having a task type is allowed to be the type of an in formal parameter of a generic unit (RM 12.1.1/3).

Test Objectives and Design Guidelines

T1. Check that task objects and objects having a subcomponent of a task type cannot be compared for equality (see IG 4.5.2/T7) or assigned.

Implementation Guideline: Include the use of assignment in an object declaration, in a default expression of a record component declaration, and in an assignment statement.

T2. Check that a component, X, of a record having a component of a task type can be assigned to if X's type is not limited.

T3. Check that a task can be passed as an actual in or in out parameter in a subprogram call.

Check that a task parameter can have a default expression.

Implementation Guideline: Include cases with composite types, and check that the formal and actual parameter denote the same tasks, i.e., check that formal and actual tasks have the same internal state.

T4. Check that a formal parameter of mode out must not have a task type or a composite type with a subcomponent of a task type (see IG 7.4.4/T1).

T5. Check that a task object declaration creates a task object prior to its activation.

Implementation Guideline: Check using the 'STORAGE_SIZE attribute, which should return a well-defined value, although the value may be zero.

T6. Check that task objects designated by access values can be interchanged by exchanging the access values.

9.3 Task Execution -- Task Activation

Semantic Ramifications

S1. The activation of a task is different from the elaboration of an object declaration that declares a task:

```
task type T;
OBJ : T;                                -- (1)
```

When the object declaration is elaborated, a task of type T is created but not activated. Because the task is not activated at (1), it doesn't matter whether T's body has been elaborated. In particular, no exception should be raised when the object declaration is elaborated; RM 3.9/6 only applies to the *activation* of tasks, not to their creation.

S2. To further emphasize the difference between elaborating an object declaration and activating a task, consider the next example. It contains a function CHECK(I) that updates a global string variable with the value of I. For example, when a call to CHECK(1) is followed by CHECK(2), the string "12" is produced. CHECK is used to see whether declarations are elaborated in the *required order*:

```
task type T;

OBJ : T;                                -- (1)
I   : INTEGER := CHECK(1);

task body T is
  K : INTEGER := CHECK(3);
begin
  K := CHECK(4);
end T;

package P is
  J : INTEGER := CHECK(2);
end P;
```

The required sequence of calls to CHECK in the above example will produce the string "1234". In particular, the declaration of T.K must not be elaborated until OBJ is being activated, which can only occur after elaborating the whole declarative part.

S3. If some components (or subcomponents) of a record type have task types, any default initializations for the nontask components are performed before the task components are activated, since RM 9.3/2 defers the activation of such tasks until the entire declarative part has been elaborated:

```
task type T;

type INNER_REC is
  record
    T1 : T;
    A  : INTEGER := CHECK(1);
  end record;

type OUTER_REC is
  record
    T2 : T;
```

```

        R : INNER_REC;
        B : INTEGER := CHECK(2);
    end record;

    task body T is
        K : INTEGER := CHECK(3);
    begin
        null;
    end T;

    package P is
        R : OUTER_REC;
    end P;

```

The string produced by the CHECK calls must be either "1233" or "2133" since all nontask component initializations must be evaluated before any tasks are activated. Similarly, tasks created by allocators are activated last:

```

type ACC_REC is access OUTER_REC;
M : INTEGER := CHECK (0);
ACC : ACC_REC := new OUTER_REC;
N : INTEGER := CHECK (4);

```

The initialization of components ACC.R.A and ACC.B occur before ACC.R.T1 and ACC.T2 are activated, so the sequence of calls to CHECK is "012334" or "021334".

S4. If an exception is raised in a declarative part before a task created by an object declaration is activated, such tasks become terminated, allowing the master containing the task objects to be left (see RM 9.4/6 and IG 9.4/3):

```

declare
    task type T;

    OBJ : T;
    I : INTEGER := 1/0;    -- NUMERIC_ERROR raised

    task body T is ... end T;
begin ... end;

```

The task designated by OBJ is terminated before it was activated. NUMERIC_ERROR is propagated from the block containing these declarations. TASKING_ERROR is not raised even though OBJ was not activated successfully.

S5. A similar situation can arise when tasks are created by allocators:

```

task type T;

type REC is
    record
        A : INTEGER := CHECK(1);
        B : T;
        C : FLOAT range -1.0 .. 1.0 := FLOAT(CHECK(2));
    end record;

type ACC_REC is access REC;

```

```

task body T is
    K : INTEGER := CHECK(3);
begin
    null;
end T;

package P is
    R : ACC_REC := new REC;  -- CONSTRAINT_ERROR raised
end P;

```

The global string produced by CHECK should have the value "12" or "21"; in no case should the string contain the value 3, since the task designated by R.B is never activated.

s6. RM 9.3/4 speaks of tasks created "indirectly" by elaborating a declarative part. Indirect creation can occur as follows:

```

package P is
    task T;
end P;
-- raise exception here
package body P is
    task body T is
        ...
    end T;
end P;

```

P.T is a task created indirectly by the elaboration of the enclosing declarative part. P.T is terminated because of the exception raised in the enclosing declarative part.

s7. When an exception is raised while activating a task, *TASKING_ERROR* is to be raised within the frame (RM 11.2/3) that is causing the task to be activated. Several tasks can be ready for activation at once (e.g., when activating or allocating an array of tasks), and more than one of these tasks may raise *TASKING_ERROR*. However, only one *TASKING_ERROR* exception is raised in the frame.

s8. If a library package P declares a task object and does not require a package body, then an implicit body is provided by RM 9.3/5. This implicit body must be elaborated prior to executing the main program (RM 10.5/1), and so all tasks declared by library units are activated before execution of the main program begins. Similarly, if pragma ELABORATE names P, then P's implicit body must be elaborated before elaborating the unit specifying the pragma, and so the task declared by P will be activated (see IG 10.5/S).

s9. Whenever a task becomes completed, the exception *TASKING_ERROR* must be raised in all tasks awaiting a rendezvous with that task (RM 11.5/2). In particular, *TASKING_ERROR* must be raised in tasks expecting a rendezvous with a task that does not become activated:

```

declare
    task type T1 is
        entry E;
    end T1;

    NOT_ACTIVE : T1;  -- will be called when not active

    task type T2;
    type ACC_T2 is access T2;  -- will call NOT_ACTIVE.E

```

```

task body T1 is
begin
    accept E;
end T1;

task body T2 is
begin
    NOT_ACTIVE.E;                -- call entry of inactive task
    FAILED ("no TASKING_ERROR raised");
exception
    when TASKING_ERROR => null;   -- okay
end T2;

package P is
    T2_OBJ      : ACC_T2 := new T2;    -- T2 now activated
    RAISE_EXCP  : INTEGER := 1/0;      -- raise NUMERIC_ERROR
end P;
begin ... end;

```

When package P is elaborated, a task of type T2 is activated and waits for its call to NOT_ACTIVE to be accepted. Elaboration of the RAISE_EXCP declaration causes NUMERIC_ERROR to be raised, so the task NOT_ACTIVE, which was waiting to be activated, becomes terminated. TASKING_ERROR must be raised in T2_OBJ.all, which allows this task to become completed. Since T2_OBJ.all and NOT_ACTIVE are both terminated, the block containing these declarations can now be left (RM 9.4/6) by propagating the NUMERIC_ERROR exception. Note that NUMERIC_ERROR is propagated (not TASKING_ERROR) because RM 9.3/3 only says TASKING_ERROR is raised when some task raises an exception while being activated, and no exception has been raised during task activation in the above example.

S10. The above example can be modified by replacing package P with the following:

```

function ABORT_NOT_ACTIVE return INTEGER is
begin
    abort NOT_ACTIVE;
    return 3;
end ABORT_NOT_ACTIVE;

package Q is
    T2_OBJ      : ACC_T2 := new T2;    -- T2 now activated
    DO_ABORT    : INTEGER := ABORT_NOT_ACTIVE; -- NOT_ACTIVE now aborted
end Q;

```

T2_OBJ.all completes its execution because TASKING_ERROR is raised within it. Execution of the block containing these declarations continues. When execution reaches the begin, no attempt is made to activate the task NOT_ACTIVE since this task is terminated. If no other tasks are created within the block, the block can be exited when it complete its execution, since all its dependent tasks are terminated.

S11. It is possible to create and activate tasks that cannot be accessed:

```

declare
    task type T;
    OBJ : array (1..2) of T;

```

```

type REC is
  record
    A    : INTEGER := CHECK(1);
    B, C : T;
  end record;
type ACC_REC is access REC;
R_OBJ : ACC_REC;
task body T is
  ...
begin ... end T;
begin
  R_OBJ := new REC;
exception
  when TASKING_ERROR =>
    abort OBJ(1), OBJ(2), R_OBJ.B, R_OBJ.C;
end;
```

Suppose that when evaluating the allocator, the task created for component B is successfully activated, but the attempt to activate the task for component C fails because an exception is raised in T's declarative part. TASKING_ERROR is raised at the point of the allocator (RM 9.3/7). However, even though a REC object has been allocated (component A has received a value), the execution of the allocator has not been successful, and so R_OBJ has not received a new value. Consequently, R_OBJ = null and evaluation of R_OBJ.B will raise CONSTRAINT_ERROR when the abort statement is evaluated. This means the task associated with component B is executing, and there is no way to name it to abort it. The block cannot be left until this task terminates its execution. Similarly, if activation of OBJ(1) is successful, but OBJ(2)'s activation fails, the task designated by OBJ(1) will not be terminated. However, in this case, since the tasks designated by OBJ(1) and OBJ(2) were created when the object declaration was elaborated, abort OBJ(1) is meaningful and will terminate the execution of OBJ(1). If OBJ(1) were not aborted, then the block could not be exited until OBJ(1) completes its execution and terminates.

S12. According to RM 3.9/6, PROGRAM_ERROR is raised by an attempt to access a task's body before it has been elaborated. Such an attempt, however, can only occur during an attempt to activate a task, and RM 9.3/3 says that if an exception is raised "by the activation" of a task, then TASKING_ERROR is raised instead of the exception that actually occurred during the activation attempt. Consequently, an attempt to access a task's body before it is elaborated raises TASKING_ERROR instead of PROGRAM_ERROR:

```

task type T;

package P is
  OBJ : T;
end P;

package body P is
begin
  null;          -- TASKING_ERROR raised here
exception
  when TASKING_ERROR => null;
  when PROGRAM_ERROR => null;  -- cannot occur
end P;

task body T is ... end T;
```

An attempt to activate P.OBJ must be made before the null statement in P is executed. Since T's body has not yet been elaborated, PROGRAM_ERROR is raised by the attempt to activate P.OBJ, but RM 9.3/3 says TASKING_ERROR is the exception that is actually propagated to the handler.

S13. Attempting to activate a task before its body has been elaborated occurs more easily for access types:

```
task type T;
type ACC_T is access T;
OBJ : ACC_T := new T;      -- TASKING_ERROR raised, not PROGRAM_ERROR
task body T is ... end T;
```

Changes from July 1982

S14. It is explicitly stated that when implicit package bodies are created for packages containing declarations of task objects, the order of elaboration of these bodies is undefined.

Changes from July 1980

S15. Activation of tasks is explicitly allowed to occur in parallel.

S16. If an exception is raised while attempting to activate one of several tasks, the remaining tasks are not affected and can be activated.

S17. When implicit package bodies are created for packages containing declarations of task objects, the bodies are placed at the end of the declarative part (not just after the associated task body).

S18. If an exception occurs while elaborating a task's declarative part, the exception is not passed to the activating task; instead, TASKING_ERROR is raised.

S19. If a record contains some components with default initializations and some components having a task type, the nontask components are initialized before any attempt is made to activate any of the task components.

Exception Conditions

E1. If elaborating the declarative part of a task causes an exception to be raised, or if an attempt is made to activate a task before its body has been elaborated, TASKING_ERROR is raised in the task that caused the attempted activation to occur. The exception is raised at the point where the activation is attempted.

Test Objectives and Design Guidelines

T1 Check that declared task objects are not activated before the end of the declarative part. (For tasks declared in package specifications, check that the tasks are not activated before the end of the package body's declarative part.)

Implementation Guideline: Check for single task objects and composite objects containing components having a task type.

Implementation Guideline: Use a mixture of blocks, subprograms, packages, and task bodies as examples of declarative parts.

Implementation Guideline: For composite objects with nontask components, check that the nontask components are initialized when the object declaration is elaborated. Try a case where subcomponents are also a mixture of task and nontask types.

T2 Check that declared task objects are activated before execution of the first statement following the declarative part.

Implementation Guideline: Check for single tasks and composite objects containing components having a task type.

Implementation Guideline: Use a mixture of blocks, subprograms, packages, and task bodies as examples of declarative parts.

- T3. Check that tasks created by allocators evaluated in a declarative part are activated when the allocator is evaluated.

Implementation Guideline: Check for single task objects and composite objects containing components having a task type.

For composite objects with nontask components, check that the nontask components are initialized first when the object declaration is elaborated.

- T4. Check that if an exception is raised when a task's declarative part is elaborated, the task is completed, and TASKING_ERROR is raised in the unit causing the task activation; the activation of other tasks should not be affected.

Check that TASKING_ERROR is not propagated until the activation of all tasks has been attempted.

Implementation Guideline: Check both for tasks created by allocators and tasks created by object declarations.

Check that when more than one task raises an exception during activation, only one TASKING_ERROR is raised.

Check that tasks waiting to rendezvous with tasks that fail to be activated receive TASKING_ERROR.

Implementation Guideline: This check can only be done for tasks created by object declarations.

- T5. Check that if an exception is raised in a declarative part, a task declared in the same declarative part becomes completed before it has been activated; no TASKING_ERROR is raised within the declarative part being elaborated, but TASKING_ERROR is raised in tasks that were awaiting a rendezvous with those tasks that become completed before being activated.

Implementation Guideline: Check when both one and several tasks are waiting to be activated when the exception is raised.

- T6. Check that a task object declared in a library package specification is activated before executing the main program, even if the package has no body.

Check that when pragma ELABORATE is applied to a package that declares a task object but has no package body, the task is activated before elaborating the unit containing the ELABORATE pragma (see IG 10.5/T4).

- T7. Check that if an attempt is made to activate a task before its body has been elaborated, the task is completed and PROGRAM_ERROR (rather than TASKING_ERROR) is raised (AI-00149).

Implementation Guideline: Check for tasks created by allocators and tasks created by object declarations.

- T8. Check that execution does not proceed in parallel with the activation of tasks.

Implementation Guideline: Check for tasks created by both object declarations and allocators.

Implementation Guideline: To maximize the chance of such an error being detected, the tasks being activated should execute a delay statement during their activation.

9.4 Task Dependence -- Termination of Tasks

Semantic Ramifications

S1. An object of a limited private type might be, or might contain, a task. Consequently, the unit in which the object is declared cannot complete its execution until the task is completed. In particular, a generic unit might be instantiated with a task type:

```
generic
  type T is limited private;
```

```
procedure P is
  X : T;
  ...
```

If P is instantiated with a task type, a call to the instantiated procedure cannot be completed until X has completed its execution.

S2. The dependence relations implied by a generic instantiation can be even more complex:

```
generic
  type T is limited private;
package P is
  X : T;
  ...
```

After instantiation, X depends on the master containing the instantiation. Similarly:

```
generic
  type LP is limited private;
  type ALP is access LP;
procedure P is
  X : ALP := new LP;
  ...
```

When this unit is instantiated, the task allocated for X is dependent on the master that elaborates the access type definition associated with ALP's actual parameter. Neither the generic unit nor the instantiation is considered a master, since the formal access type definition is never elaborated (see RM 12.1/6 and RM 12.3/17).

S3. When a master is a task, dependence is not textually determined by the position of the task body, but rather by the manner in which a task object is created:

```
procedure P is
  task type TT;
  task body TT is
    task INNER;
    ...
  end TT;
  type ATT is access TT;
begin
  B1: declare
    X : TT;
    Y : ATT := new TT;
  begin
```

The task designated by X depends on block B1. This task is the master of another task called INNER, which indirectly depends on B1. Block B1 cannot be left until both X and X's dependent task have terminated. On the other hand, the task designated by Y.all depends on procedure P, and therefore, the Y.all's INNER task indirectly depends on P.

S4. For an access type, dependence is on the collection associated with the access type:

```

B1:  declare
      task type T;
      type A_T is access T;           -- (1)
      ...
    begin
B2:  declare
      type DA_T is new A_T;           -- (2)
      X : DA_T := new T;              -- depends on B1
      Y : A_T  := new T;              -- depends on B1

```

X depends on B1 because the declaration of DA_T does not contain an access type definition. The only access type definition related to DA_T is the definition used at (1) to declare A_T. RM 9.4/2 defines dependence on the master that "elaborates the corresponding access type definition," and the definition at (1) is the only one related to DA_T. It is easy to misread RM 9.4/2 as saying "the corresponding access type *declaration*" (meaning the declaration at (2)).

S5. When a selective wait with a terminate alternative appears in an inner block that is a master, the implementation must be careful to evaluate termination conditions correctly:

```

task body T is
  T1, T2 : TASKS;
begin
B:  declare
      Ta, Tb : MORE_TASKS;
    begin
      ...
      -- (1) selective wait with terminate here
      ...
    end B;          -- Ta, Tb must be terminated
end T;              -- T1, T2 must be terminated

```

Execution must be suspended at the end of block B until Ta and Tb terminate. If these tasks are executing a selective wait with an open terminate alternative, the terminate alternative can be taken. If execution is waiting at (1), then the terminate alternative can be taken only if task T is able to be terminated, i.e., if all of T's dependent tasks are either terminated or waiting at an open terminate alternative. In particular, the terminate alternative can only be taken if Ta, Tb, T1, and T2 are all terminated or are able to select terminate alternatives.

S6. The idea behind the terminate alternative is that it is selected when no more work can be done. But this does not mean the terminate alternative can be selected when tasks are deadlocked. For example, a terminate alternative cannot be selected when a task is queued for an entry of a task executing a selective wait with terminate, even if the entry has no accept statements or is in a closed accept alternative. If such an entry has been called with a timed entry call, then when the delay has expired, the entry is removed from the queue, and the conditions for termination may be met if the calling task subsequently terminates.

S7. The definition of completion for a block should have mentioned that a block is completed if an exception is raised by the elaboration of its declarative part, since this is one way of leaving a block. In particular, if a task is dependent on a block, and the elaboration of the block's declarative part raises an exception, the block is not left until the dependent task terminates:

```

declare
  type ACC_T is access SOME_TASK_TYPE;
  LOST_TASK : ACC_T := new SOME_TASK_TYPE;
  I : NATURAL := -1;          -- CONSTRAINT_ERROR raised
begin

```

LOST_TASK.all depends on the block, since the block elaborates the access type definition for ACC_T. CONSTRAINT_ERROR cannot be propagated from the block until LOST_TASK.all has terminated.

S8. Now consider a case where a task object is declared in a block that raises an exception while elaborating its declarative part:

```

declare
  OBJ : SOME_TASK_TYPE;
  I   : NATURAL := -1;           -- CONSTRAINT_ERROR raised
begin

```

RM 9.3/4 applies here: the task designated by OBJ becomes terminated without being activated, and the block can be left immediately.

S9. A task object designated by an object of an access type may become inaccessible before it terminates, e.g.:

```

declare
  task type T is ... end T;
  task body T is ... end T;
  type T_ACC is access T;
begin
  ...
  declare
    MORIBUND: T_ACC := new T;
  begin
    null;
  end;
  -- now we've done it!
end;

```

Completion of the inner block does not await completion of the newly created task object. Although the object designated by MORIBUND is not accessible after execution has left the block, the storage allocated for the task designated by MORIBUND.all cannot be reclaimed until MORIBUND.all is terminated (RM 4.8/7). Moreover, the outer block cannot be left until MORIBUND.all terminates because MORIBUND.all depends on the outer block; it is not relevant whether a dependent task can be accessed by some name (see also IG 9.3/S).

S10. If a task object is declared in a library package, or if an access to task type is declared in a library package specification, the library package serves as a master (RM 9.4/1). The RM allows a main program to complete its execution even though some tasks dependent on a library package have not completed their execution. In any event, tasks dependent on a library package must not be aborted when the main program is completed (in the sense of RM 9.4/5); they must be allowed to continue to run until they terminate. Some Ada applications, for example, might have a null main program, with all the real work being done by tasks dependent on library packages. In real-time control systems, such tasks might never be expected to terminate and their activation would be the equivalent of system startup.

S11. Since completion is not defined for a library package, a master that is a library package can never be said to be complete. Consequently, if all tasks dependent on library packages are waiting on open terminate alternatives of select statements and the main program is complete, no rule requires that all such tasks be terminated. However, since no further execution is possible, the tasks might as well be considered terminated, e.g., in a batch job environment, the next job can begin execution at this time. If a task dependent on a library package is neither complete nor waiting on an open terminate alternative when the main program completes, then the task must be allowed to complete its execution.

S12. The rules allow a task to be accessed from outside its master, even though in such a case, the task must have terminated:

```

declare
  task type T;
  type ACC_T is access T;

  function F return T is
    LOCAL_TASK : T;
  begin
    return LOCAL_TASK;
  end F;

  task body T is ... end T;
begin
  if F'TERMINATED then           -- must be true

```

The value returned by F can be used as the prefix of the 'TERMINATED attribute. Since F cannot return until LOCAL_TASK is terminated, F'TERMINATED will always yield TRUE. This example shows that under some circumstances, even after a master is exited, there can be further references to a dependent task created within the master.

S13. When a select statement contains both a terminate alternative and an accept alternative for an entry associated with a hardware interrupt, an implementation is allowed to impose further requirements for the selection of the terminate alternative in addition to those given in RM 9.4/7-10 (see RM 13.5.1/3).

Changes from July 1982

S14. Indirect dependence is defined for subprograms.

S15. When a master is a task, the task does not depend on the master that activates the task; instead the usual rules for dependence are used.

Changes from July 1980

S16. For access types, incorrect wording indicating dependence on package bodies has been replaced with wording indicating dependence on task bodies.

S17. The wording regarding dependence has been clarified by introducing the concept of a "master."

S18. An explicit definition is given of the circumstances under which the execution of a construct is considered complete.

S19. Task completion is distinguished from task termination so the rules explaining termination of dependent tasks can be expressed more clearly and correctly.

Test Objectives and Design Guidelines

T1. Check that a unit with dependent tasks created by object declarations is not completed until all dependent tasks become terminated.

Implementation Guideline: Do not use the terminate alternative; this will be checked separately in T8.

Implementation Guideline: Include objects having a limited private type whose full declaration declares a task type or a type having a component of a task type.

Implementation Guideline: Include some indirect dependencies.

Implementation Guideline: Check that a unit is not exited until all dependent tasks are terminated even if an attempt is made to leave the unit by raising an exception.

- T2. Check that a unit that creates tasks by allocators completes prior to termination of the allocated task if the unit did not declare the access type; otherwise, it waits until the created tasks have all terminated.

Implementation Guideline: To ensure that the set of dependent tasks is not determined statically, create a linked list of records containing tasks.

Implementation Guideline: Use a limited private type and a composite type with task subcomponents.

Check that the master for a derived access type is the unit containing the access type definition.

Implementation Guideline: Use two levels of derivation to ensure the master is not merely considered to be the parent of the derived type.

- T3. Check that a unit terminates properly if it declares an access type designating task objects but never actually creates a task.

- T4. Check that tasks dependent on a library package continue to execute even after completion of the main program.

Implementation Guideline: Include task objects declared in a library package specification or body and access to task types, where the tasks are activated either in the package body or in the main program.

Implementation Guideline: Make the main program of high priority and the tasks of lower priority. The tasks should contain a delay statement to allow the main program to complete, although there is no way to ensure that the main program completes.

Implementation Guideline: Check separately whether such tasks are terminated when all of them are waiting at open alternatives of a select statement.

- T5. Check that if a task type is declared in a library package, a main program that declares objects of that type waits for the termination of such objects.

- T6. Check that a declaration that renames a task does not create a new master for the task.

Check that a subtype declaration for an access type does not create a new master with respect to the unit containing the subtype declaration.

- T7. Check that a task object declared in a nonlibrary package does not depend on the package, but does depend on the innermost enclosing block, subprogram, or task body, i.e., the elaboration of declarations following the package body can continue even if the task is not yet terminated.

Implementation Guideline: Include a check for leaving a package body by raising an exception. Include package body subunits in this check.

- T8. If tasks directly or indirectly dependent on a master are all either terminated or are executing a select statement with an open terminate alternative, check that the master is completed.

Implementation Guideline: Check that the master is not terminated if only some tasks are at an open terminate alternative.

Implementation Guideline: Include a case where a terminate alternative is inside a block that has dependent tasks (see example in IG 9.4/S above).

- T10. If a generic unit has a formal limited private type and declares an object of that type (or has a subcomponent of that type), and if the unit is instantiated with a task type or an object having a subcomponent of a task type, check that the usual rules apply to the instantiated unit, namely:

- if the generic unit is a subprogram, control cannot leave the subprogram until the task created by the object declaration is terminated;

Implementation Guideline: Include a check where the subprogram is to be left by raising an exception.

- if the generic unit is a package, control cannot leave the enclosing master until the task created by the object declaration is terminated.

Implementation Guideline: Include a case where the actual parameter of the instantiation is a limited private type whose full declaration declares a task type.

Check that if the generic unit declares an access type whose designated type is a formal limited private type, then the appropriate termination rules apply when the unit is instantiated with a task type or a type having a component of a task type.

- T11. Check that if a formal access type of a generic unit designates a formal limited private type, then when the unit is instantiated with a task type or a type having a subcomponent of a task type, the master for any tasks allocated within the instantiated unit is the master determined by the actual parameter.
- T20. Check that if the completion of one task will allow a master to be completed, then the task can be completed by aborting it, and the master will continue its execution.
- T21. Check that a task can be accessed from outside its master.

9.5 Entries, Entry Calls, and Accept Statements

Semantic Ramifications

S1. The scope of a formal parameter in an entry declaration starts at the beginning of the parameter's declaration (RM 8.2/2). Hence, the identifier of a formal parameter can be used in an entry declaration prior to the formal parameter's declaration as long as it denotes an entity other than the formal parameter:

```
C : constant := 5;
task type T is
    entry E (1..C) (X : INTEGER := C; C : FLOAT);
end T;
```

The above uses of C are legal and refer to the constant declared outside the task type. The identifier C cannot, however, be used within the formal part to denote a preceding formal parameter (see IG 6.1/S and RM 6.1/5).

S2. An entry may be called by the main program as well as from another task since a main program is considered to be called from a task (see RM 10.1/8).

S3. If the discrete range used to declare an entry family has the form L .. R and each bound has type *universal_integer* and is either a numeric literal, named number, or attribute, L and R are implicitly converted to the predefined type INTEGER (see RM 3.6.1/2 and IG 3.6.1.a/S).

S4. Control can leave a rendezvous by executing a return statement (see RM 5.8/1), as well as by raising an exception or simply by normally completing the execution of the sequence of statements contained by the accept statement. Control cannot leave by executing an exit statement (see RM 5.7/3) or a goto statement (see RM 5.9/3).

S5. If an entry has a parameter of mode in out or out, assignments to the formal parameter can immediately affect the value of the actual parameter if the parameter is passed by reference. In such a case, the value of the actual parameter is updated during the rendezvous. If the parameter is passed by copy, however, the RM 9.5/14 applies: "[After the accept statement's sequence of statements has been executed], the calling task and the task owning the entry continue their execution in parallel." Since the execution of the accept statement's statements says nothing about the treatment of parameters, this wording implies that any copy back action occurs after completion of the rendezvous, in the context of the call. Of course, an implementation must ensure that the formal parameters are not replaced with new values associated with the next rendezvous until the old values are safely transmitted to the calling

task. For example, if the calling task is running on another computer, parameter results can be given to the calling task, which then is responsible for copying them to the actual parameters while the called task can proceed to rendezvous with another task.

S6. An entry can be declared even though there is no accept statement for it in the body of the task. Of course, if such an entry is called, it will never be accepted, and the calling task will eventually receive TASKING_ERROR when the called task terminates.

S7. Evaluation of an entry name is described in RM 4.1/10, RM 4.1.1/3, and RM 4.1.3/9-10. In particular, if the prefix denotes an access value, no .all is needed:

```
task T is
    entry E;
end T;

type A_T is access T;
X : A_T;
... X.E;           -- legal entry call
... X.all.E;       -- also legal
```

S8. Interrupts are treated as entry calls (RM 13.5.1); an interrupt entry is not executed (in response to the corresponding interrupt) unless a corresponding accept statement is executed. Although entries can be associated with hardware interrupts (RM 13.5/6), such entries can still be called directly with entry calls.

S9. An entry can be overloaded either by using the same identifier to declare other entries of the same task or an enclosing task, or to declare subprograms or enumeration literals in the task body or in an enclosing unit.

S10. An exception raised inside an accept statement and not handled locally is propagated both to the unit containing the accept statement and to the calling task (RM 11.5/4). If the called task is aborted during a rendezvous, TASKING_ERROR is raised in the calling task at the place of call (RM 11.5/5); the rendezvous need not be completed (see RM 9.10/7). On the other hand, if a calling task is aborted while in a rendezvous, the rendezvous must be completed before the calling task is terminated (see RM 9.10/6). No exception is raised in the called task (see RM 11.5/6).

S11. When evaluating a call to a member of an entry family, the entry family index is evaluated before any actual parameters (RM 9.5/10).

S12. Since an entry family name is neither an object nor a type, the definition of SUPPRESS does not permit an entry family to be named in a SUPPRESS pragma. Consequently, the only way to specify that index checks are to be suppressed when calling or accepting a call for a member of an entry family is to name the index type in the pragma.

S13. Most of the rules applicable to subprogram declarations and calls also apply to entry declarations and calls. These rules are repeated in this section, and tests to check that subprogram calls are performed correctly are repeated for entry calls (see T61-T95).

Changes from July 1982

S14. There are no significant changes.

Changes from July 1980

S15. The syntax of entry calls and accept statements explicitly allows an entry family index to be declared and specified.

S16. The name of a single entry or entry family in an accept statement is restricted to a simple name; no expanded names can be used.

- S17. Additional restrictions have been imposed on the use of a task name within its body.
- S18. The rule restricting the use of accept statements within a task body has been augmented to ensure that a task can execute accept statements only for its own entries.
- S19. Overloading is disallowed for entry family identifiers.
- S20. The evaluation of an entry call now explicitly requires evaluation of the entry name, not just the entry index.
- S21. `TASKING_ERROR` is raised if the called task completes execution before accepting the call.

Legality Rules

- L1. If a name of an entry family is given in either an accept statement or an entry call, it must be followed by an index expression (RM 9.5/3, /4, /7, /10), and the base type of the index in the name must be the same as the base type of the entry index given in the entry family declaration (RM 9.5/4).
- L2. The name of an entry family cannot be overloaded (RM 9.5/5).
- L3. The name of a single entry can be overloaded (RM 9.5/5), i.e., two single entries or a single entry and a procedure having the same identifier can be declared in the same declarative region if the number, order, and base types of the parameters are not all the same (RM 8.3/15, /17).
- L4. The formal part of an accept statement must conform to the formal part given in the declaration of the single entry or entry family named by the accept statement (RM 9.5/7), i.e.,
- the formal parts must consist of the same sequence of lexical elements, except that comments are ignored, certain string literals can be replaced by different string literals (see below), and simple names can be replaced by expanded names if the meaning of both names is given by the same declaration (RM 6.3.1/5).
 - corresponding numeric literals must have the same (*universal_integer* or *universal_real*) value (RM 6.3.1/2).
 - a character literal or an operator symbol cannot be replaced by an expanded name denoting the same literal or operator (RM 6.3.1/3).
 - corresponding string literals used as operator symbols can differ only with respect to the case of the letters used in the operator symbol (RM 6.3.1/4).
 - corresponding simple names, character literals, operators, and operator symbols must be declared by the same declaration (RM 6.3.1/5).
- L5. A simple name appearing at the end of an accept statement must match the single entry or entry family name given at the beginning (RM 9.5/7).
- L6. An accept statement for an entry of a given task must occur within the corresponding task body, but not within the body of any task, package, or subprogram that is itself declared within the task (RM 9.5/8).
- L7. An accept statement for an entry of a given task must not occur within another accept statement for either the same single entry or for an entry of the same family (RM 9.5/8).
- The remaining rules derive from the rules for subprogram declarations as applied to entry declarations, in accordance with RM 9.5/6.

- L8. In entry declarations, a default expression is allowed only for formal parameters having mode in (RM 9.5/6 and RM 6.1/4).
- L9. The base type of a default expression must be the same as the base type of its formal parameter (RM 9.5/6 and RM 6.1/4).
- L10. A simple name is not allowed in a parameter declaration if the name denotes a formal parameter declared earlier in the same formal part (RM 9.5/6 and RM 6.1/5).
- L11. The identifier declared as the name of an entry or an entry family cannot be used within the entry's formal part except to declare a formal parameter having the same identifier (RM 8.3/16). In particular, its use as a selector in a component selection, as a component simple name in an aggregate, as a parameter name in a named parameter association, or as a simple name in a default expression is forbidden.
- L12. The formal parameters of an entry must have identifiers that are distinct from each other and from identifiers declared in the task body's declarative part (RM 8.3/17).
- L13. An out parameter of an entry declaration must not have a limited type unless (RM 7.4.4/4):
- the type is a limited private type,
 - the declaration of the entry occurs within the visible part of the package that declares the limited private type (including within any nested packages), and
 - the full declaration of the limited private type does not declare a limited type.
- L14. A formal in parameter or a subcomponent of a formal in parameter of an entry must not be used as an actual in out parameter of an entry or a subprogram call (RM 9.5/6 and RM 6.4.1/3), as an actual out parameter (RM 9.5/6 and RM 6.4.1/3), as the target of an assignment statement (RM 5.2/1), or as a generic in out actual parameter (RM 12.3.1/2).
- L15. A formal out parameter or a subcomponent of a formal out parameter must not be used as an actual in out parameter of a subprogram call or an entry call (RM 9.5/6 and RM 6.4.1/3), or as an actual in out parameter in a generic instantiation (RM 12.3.1/2).
- L16. A formal out parameter or a subcomponent of a formal out parameter (other than a discriminant subcomponent) must not be used in an expression (RM 9.5/6 and RM 6.2/5), except as the prefix of the attribute ADDRESS, CONSTRAINED, FIRST and FIRST(N) (when the parameter has an array type), FIRST_BIT, LAST and LAST(N) (when the parameter has an array type), LAST_BIT, LENGTH and LENGTH(N) (when the parameter has an array type), POSITION, RANGE and RANGE(N) (when the parameter has an array type), or SIZE.
- L17. The prefix of an attribute cannot be an out parameter or a subcomponent of an out parameter if the parameter or subcomponent has an access type (RM 4.1/4).
- L18. For an entry call with only positional parameters:
- the number of actual parameters must equal the number of formal parameters (RM 9.5/6 and RM 6.4/5); or
 - the number of actual parameters must be less than the number of formal parameters, and, if N parameters are omitted, the last N formal parameters must have default values specified for them (RM 9.5/6 and RM 6.4/5);
 - the base type of the *i*th actual and formal parameter must be the same (RM 9.5/6 and RM 6.4.1/1).
- L19. For an entry call with both named and positional parameters,

- the total number of actual parameters must not exceed the number of formal parameters (RM 9.5/6 and RM 6.4/5);
 - omitted actual parameters must correspond to formal parameters for which default values were specified (RM 9.5/6 and RM 6.4/5);
 - positional parameters must appear first (RM 9.5/6 and RM 6.4/4);
 - a named parameter must not be specified for a formal parameter if an actual positional parameter is also given for that formal parameter (RM 9.5/6 and RM 6.4/5);
 - the base types of corresponding formal and actual parameters must be the same (RM 9.5/6 and RM 6.4.1/1).
- L20. The formal parameter name in a named parameter association must be identical to that of a formal parameter in the corresponding entry declaration (RM 9.5/6 and RM 6.4/3).
- L21. No duplicates are permitted among the formal parameter names used in any parameter associations of an entry call (RM 9.5/6 and RM 6.4/5).
- L22. An actual out or in out parameter of an entry call must be either the name of a variable or must have the form of a type conversion applied to the name of a variable (RM 9.4/6 and RM 6.4.1/3).
- L23. The type mark appearing in an actual in out or out parameter having the form of a type conversion must conform to the type mark of the formal parameter (RM 9.4/6 and RM 6.4.1/3).
- L24. A call to a single entry is not allowed unless the name of the entry, the number of parameter associations, the types and order of the actual parameters, and the names of the formal parameters (if named associations are used) suffice to determine which entry is being called (RM 9.4/6 and RM 6.6/3).

Exception Conditions

- E1. For an entry call, `TASKING_ERROR` is raised if the called task:
- has completed (or terminated) its execution at the time of the call (RM 9.5/16); or
 - completes its execution before accepting the call (RM 9.5/16); or
 - is abnormal when the call is made (i.e., the task has been aborted, but is not yet terminated) (RM 9.10/7); or
 - becomes abnormal (i.e., is aborted) before accepting the call or while executing the rendezvous (RM 9.10/7).
- E2. `CONSTRAINT_ERROR` is raised for an entry call if the value of the entry index does not lie within the declared range (RM 9.5/16).
- E3. `CONSTRAINT_ERROR` is raised for an accept statement if the value of the entry index does not lie within the declared range for the entry family (RM 9.5/16).
- E4. For an actual parameter of mode in, `CONSTRAINT_ERROR` is raised if:
- the formal parameter has a scalar type and the value of the actual parameter before the call lies outside the range specified for the formal parameter (RM 9.5/6 and RM 6.4.1/6).

- the formal parameter has a constrained array type and the index bounds of the actual parameter are not equal to the bounds specified for the formal parameter (RM 9.5/6 and RM 6.4.1/6, /9).
- the formal parameter has a constrained record, private, or limited private type and the discriminant values for the actual parameter do not equal those specified for the formal parameter (RM 9.5/6 and RM 6.4.1/6, /9).
- the formal parameter has a constrained access type, the value of the actual parameter is not null, and the bounds or discriminants of the designated object do not equal the values of the bounds or discriminants specified for the formal parameter's constraint (RM 9.5/6 and RM 6.4.1/6).

E5. For a parameter of mode In out having the form of a variable name,

- **CONSTRAINT_ERROR** is raised before the call if:
 - the parameter has a scalar type and the value of the variable lies outside the range specified for the formal parameter (RM 9.5/6 and RM 6.4.1/6).
 - the formal parameter has a constrained array type and the index bounds of the actual parameter are not equal to the bounds specified for the formal parameter (RM 9.5/6 and RM 6.4.1/6, /9).
 - the formal parameter has a constrained record, private, or limited private type, and the discriminant values for the variable do not equal those specified for the formal parameter (RM 9.5/6 and RM 6.4.1/9, /6).
 - the formal parameter is a constrained access type, the value of the variable is not null, and the bounds or discriminants of the designated object do not equal the values of the bounds or discriminants specified for the formal parameter's constraint (RM 9.5/6 and RM 6.4.1/6).
- **CONSTRAINT_ERROR** is raised after the completion of the entry call if:
 - the formal parameter has a scalar type or a private type whose full declaration declares a scalar type and the value of the formal parameter lies outside the range specified for the variable named as the actual parameter (RM 9.5/6 and RM 6.4.1/7).
 - the actual parameter has a constrained access type or a private type whose full declaration declares a constrained access type, the formal parameter's value is not null, and the bounds or discriminants of the formal parameter's designated object do not equal the values of the bounds or discriminants specified for the actual variable's subtype (RM 9.5/6 and RM 6.4.1/7).

E6. For a parameter of mode In out having the form of a type conversion applied to the name of a variable:

- **NUMERIC_ERROR** is raised before the call if:
 - the parameter has a scalar numeric type and the value of the actual parameter cannot be accurately represented as a value of the formal parameter's type because the value lies outside the range of the formal parameter's base type (RM 3.5.4/10).

- the formal parameter has an unconstrained array type and for some dimension of the formal parameter's type, an index bound of the variable lies outside the range of the formal parameter's index base type (RM 9.5/6, RM 6.4.1/4, and RM 4.6/13).
- **CONSTRAINT_ERROR** is raised before the call if:
 - the parameter has a scalar type and the converted value of the variable lies within the range of the formal parameter's base type but outside the range specified for the formal parameter (RM 9.5/6, RM 6.4.1/4, and RM 4.6/12).
 - the formal parameter has an array type:
 - constraints are specified for the component type of the variable and the component type of the formal parameter, and the constraints are not equal (RM 9.5/6, RM 6.4.1/4, and RM 4.6/13); or
 - the array type is unconstrained, the operand is a non-null array, and for some dimension of the formal parameter's type, the index bounds of the variable, after conversion to the formal parameter's index base type, do not both lie within the range of the formal parameter's index subtype (RM 9.5/6, RM 6.4.1/4, and RM 4.6/13).
 - the array type is constrained,
 - the formal parameter declares a null array, and the value of the variable is not a null array (RM 9.5/6, RM 6.4.1/4, and RM 4.6/13); or
 - the formal parameter does not declare a null array, and for at least one dimension, the number of components specified for the value of the variable is not the same as the number of components specified for the formal parameter (RM 9.5/6, RM 6.4.1/4, and RM 4.6/13);
 - the formal parameter is a constrained access type, the value of the variable is not null, and the bounds or discriminants of the designated object do not equal the values of the bounds or discriminants specified for the formal parameter's constraint (RM 9.5/6, RM 6.4.1/4, and RM 4.6/12).
- **NUMERIC_ERROR** is raised after the call if the parameter has a scalar numeric type and the value of the formal parameter cannot be accurately represented as a value of the actual parameter's type because the value lies outside the range of the actual parameter's base type (RM 3.5.4/10).
- **CONSTRAINT_ERROR** is raised after the call if:
 - the parameter has a scalar type or a private type whose full declaration declares a scalar type, and the converted value of the formal parameter lies within the range of the actual parameter's base type but outside the range specified for the actual variable (RM 9.5/6 and RM 6.4.1/7).
 - the formal parameter is a constrained access type or a private type

whose full declaration declares a constrained access type; the value of the formal parameter is not null; and the bounds or discriminants of the designated object do not equal the values of the bounds or discriminants specified for the actual parameter (RM 9.5/6 and RM 6.4.1/7).

E7. For a parameter of mode out having the form of a variable name:

- **CONSTRAINT_ERROR** is raised before the call as for an in out formal parameter of a constrained array, record, private, or limited private type (see E5).
- **CONSTRAINT_ERROR** is raised after normal completion of the call as for in out parameters (see E5).

E8. For a parameter of mode out having the form of a type conversion applied to the name of a variable:

- **NUMERIC_ERROR** is raised before the call as for an in out formal parameter of an unconstrained array type (see E6).
- **CONSTRAINT_ERROR** is raised before the call as for in out formal parameters having an array type (see E6).
- **NUMERIC_ERROR** is raised after completion of the call as for an in out formal parameter of a scalar type (see E6).
- **CONSTRAINT_ERROR** is raised after completion of the call as for an in out formal parameter of a scalar, private, or constrained access type (see E6).

E9. For omitted parameter associations, **CONSTRAINT_ERROR** is raised:

- for scalar parameters if the value of the default expression lies outside the range of the formal parameter (RM 9.5/6 and RM 6.4.1/6).
- for formal parameters having a constrained array type if the bounds of the default expression do not equal the bounds specified for the formal parameter (RM 9.5/6 and RM 6.4.1/6, /9).
- for formal parameters having discriminants if the discriminants for the value of the default expression do not equal the discriminants specified for the formal parameter (RM 9.5/6 and RM 6.4.1/6, /9).
- for formal parameters of a constrained access type if the value of the default expression is not null and the bounds or discriminants of the designated object do not equal the bounds or discriminants specified for the formal parameter (RM 9.5/6 and RM 6.4.1/6).
- for formal parameters of a private type if the value of the default expression would raise an exception when the appropriate rule for the private type's full declaration is used.

Test Objectives and Design Guidelines

T1. Check that the name of an entry family must be specified as a singly indexed component in an entry call or an accept statement.

Check that the base type of an index in an entry call or an accept statement must match that given in the entry family's declaration.

Check that the base type of the index is `INTEGER` if both bounds are specified with integer literals, named integer constants, or attributes having the type *universal_integer*.

Check that a discrete range for an entry index is illegal if both bounds have type *universal_integer* but the bounds have the form of a signed integer literal, a signed attribute, or a signed named constant.

- T2. Check that the name of a single entry may not be given as an indexed component in an accept statement or in an entry call.
- T3. Check that the entry name in an accept statement cannot be an expanded name.
- T4. Check that an accept statement for an entry of a task may not appear outside the task's body, or in a subprogram, a package, or a task unit nested in the task's body.

Check that an accept statement cannot be nested with another accept statement for the same single entry or entry family (even if different entry family members are named).

Check that nested accept statements are allowed for single entries having the same name (and different parameter profiles) (see T22).
- T5. Check that the identifier at the end of an accept statement may be omitted.
- T6. Check that the identifier at the end of an accept statement, if present, must be that used in the declaration of the corresponding entry or entry family.
- T7. Check that if a task's entry is renamed as a procedure inside the corresponding task body, the procedure name must not be used in an accept statement.
- T8. Check that `CONSTRAINT_ERROR` is raised for an out-of-range index value when referencing an entry family, either in an accept statement or an entry call.
- T9. Check that a task object can call its own entries and entries of other tasks.

Implementation Guideline: Deadlock occurs if a task calls its own entry. Use a timed entry call to break the deadlock.
- T10. Check that a task may contain more than one accept statement for an entry.

Implementation Guideline: Use single entries and entry families.
- T11. Check that a task need not contain an accept statement for an entry.
- T12. Check that calling a task not yet activated does not raise an exception at the point of the call.
- T13. Check that a rendezvous in which the accept statement has no `do` part is carried out.
- T20. Check that the formal parts of entry declarations and accept statements must conform.

Implementation Guideline: Include a check across separately compiled units.

Implementation Guideline: Include a check that all entry family members have the same formal part.
- T21. Check that entry calls are processed in the order of their arrival.

Implementation Guideline: Include a check that each entry family member has its own queue.
- T22. Check that an accept statement can be executed from inside another accept statement.

Implementation Guideline: Include a case where the entries have the same identifier (but are overloaded)
- T30. Check that an entry declaration is not allowed in a task body.
- T31. Check that nonexistent entries cannot be called.
- T32. Check that selected component notation must be used to call entries from outside a task (unless the entries have been renamed as procedures).

T33. Check that the entry family index is evaluated in accept statements and entry calls.

Implementation Guideline: In the entry call case, check that the index is evaluated before any argument in the call.

T34. Check that a calling task is suspended if the receiving task has not reached the corresponding accept statement.

Check that the calling task remains suspended while the rendezvous is in progress.

T35. Check that a task is suspended when it reaches an accept statement if there is no call currently waiting for that entry.

T40. Check that TASKING_ERROR is raised if an entry of a completed task is called, or if a called task is active but completes without accepting the call.

T41. Check that an entry family index can be specified with the form, A'RANGE.

Checks based on subprogram declaration and call rules

T61. Check that certain syntactic malformations are forbidden, namely (cf. IG 6.1/T1):

- an array type definition is not allowed in a formal parameter declaration;
- the type of a formal parameter must be designated by a type mark, not a subtype indication with an explicit constraint.

T62. Check that within an entry declaration, duplicate formal parameter names are forbidden in a formal part (cf. IG 8.3.e/T1).

T63. Check that default expressions are forbidden for formal parameters of mode in out or out (cf. IG 6.1/T5).

T64. Check that the type of a default expression must be the same as the base type of the formal parameter (cf. IG 6.1/T6).

T65. Check that CONSTR.4INT_ERROR is not raised when an entry is declared if the value of the default expression for the formal parameter does not satisfy the constraints of the type mark, but is raised when the entry is called and the default value is used (cf. IG 6.1/T8).

Implementation Guideline: Try: an array parameter constrained with nonstatic bounds and initialized with a static aggregate; a scalar parameter with nonstatic range constraints initialized with a static value; and a record parameter whose components have nonstatic constraints initialized with a static aggregate.

T66. Check that names of variables, calls to user-defined operators, calls to functions, and allocators may be used in default expressions for formal parameters (cf. IG 6.1/T9).

T67. Check that a formal parameter of mode in and in out can have a limited type, including a composite type (cf. IG 6.1/T10).

Check that a formal parameter of mode out may be a limited private type only under certain circumstances (see IG 7.4.4/T1).

Check that a formal parameter of mode out cannot be a task type (see IG 7.4.4/T1).

T68. Check that a name referring to a formal parameter cannot be used later in the same formal part (although a parameter's identifier can be used if it does not refer to the parameter) (cf. IG 6.1/T11).

T69. Check that the identifier of a single entry or an entry family cannot be used within its formal part as a selector, as a component simple name in an aggregate, as a parameter name in a named association, or as a simple name in a default expression (cf. IG 6.1/T12).

T70. Check that a formal in parameter cannot be used as the target of an assignment

statement, or as an actual parameter whose mode is in out or out, nor as a generic in out actual parameter (see IG 12.3.1/T2) (cf. IG 6.2/T1).

Implementation Guideline: Use a simple scalar in parameter, an array and a record in parameter (attempt to assign to a component of the parameter or to use a component as an out actual parameter), and an in parameter having an access type.

- T71. Check that objects designated by in parameters of access types (including the object selected by .all) can be used as the target of an assignment statement and as an actual parameter of any mode (cf. IG 6.2/T2).

- T72. Check that scalar and access parameters are copied for all three modes (cf. IG 6.2/T3).

Implementation Guideline: Check this with entry calls having the form $F(A,A)$ where the second parameter is an in out or out parameter. Assignments to the second formal parameter should not change the value of the first formal parameter, nor should direct assignments to the actual parameter change the value of the corresponding formal parameter.

Implementation Guideline: Check that if an exception is propagated from a subprogram, the values of the actual scalar parameters are the values at the time of the calls, even if assignments were made to the formal parameters before the exception was raised.

Check that a private type whose full declaration declares a scalar or an access type is passed by copy for all modes (cf. IG 6.2/T3).

- T73. Check that aliasing is permitted for parameters of composite types, e.g., a matrix addition procedure can be called with three identical arguments (cf. IG 6.2/T4).

- T74. Check that the discriminant of an out formal parameter and its subcomponents may be read inside the procedure, but not other component values (cf. IG 6.2/T6).

Check that 'FIRST', 'LAST', 'LENGTH', 'RANGE', 'ADDRESS', 'SIZE', 'POSITION', 'FIRST_BIT', and 'LAST_BIT' cannot be applied to an out parameter of an access type (nor to an access subcomponent of an out parameter), but are allowed for an in or in out parameter (cf. IG 6.2/T6).

Check that 'FIRST', 'LAST', 'LENGTH', 'RANGE', 'ADDRESS', 'SIZE', 'POSITION', 'FIRST_BIT', and 'LAST_BIT' can be applied to an out parameter or an out parameter subcomponent that does not have an access type (cf. IG 6.2/T6).

Check that 'CONSTRAINED' is not allowed for a parameter (of any mode) having an access type, even if the designated type is a type with discriminants (cf. IG 6.2/T6).

Check that an out parameter or an out parameter subcomponent cannot be read or passed as an in or in out parameter (cf. IG 6.2/T6).

Check that an out parameter or an out parameter subcomponent having an access type cannot be used in a selected component, an indexed component, or a slice (cf. IG 6.2/T6).

Implementation Guideline: Check in both expression and assignment contexts.

Check that no out parameter or out parameter subcomponent can be used as an in out actual parameter (cf. IG 6.2/T6).

Check that an out parameter can be passed to another out parameter (cf. IG 6.2/T6).

- T75. Check that entry calls are allowed even if they do not assign values to scalar formal out parameters or to scalar components of formal out parameters. No exceptions should be raised, and no errors reported in compilation. (Warnings, however, are allowed) (cf. IG 6.2/T7).

Implementation Guideline: Do not refer to the values after the calls.

- T76. Check that an accept statement with and without a return statement returns correctly (cf. IG 6.3/T4).

T77. Check that no exception handler part can be provided in an accept statement.

T78. Check that an exception raised during the execution of an accept statement can be handled inside the accept body.

T79. Check that the form T.F() is illegal for an entry call (cf. IG 6.4/T1).

T80. For entries having no default parameter values, check that the number of actual positional and named parameters must equal the number of formal parameters (cf. IG 6.4/T2).

Implementation Guideline: Use calls that are valid except for the number of parameters. Check calls to single entries and to entry family members. Use purely positional notation, purely named notation, and a combination of positional and named notation.

Check that parameterless entries can be called with the appropriate notation (cf. IG 6.4/T2).

Check that the base type of formal and actual parameters must be the same (cf. IG 6.4/T2).

Implementation Guideline: Check numeric types in particular.

T81. Check that for a mixture of named and positional notation, named parameters cannot precede, or be interleaved with, positional parameters (cf. IG 6.4/T3).

Implementation Guideline: Use a call in which the types of all the formal parameters are identical. Check that in a mixture of named and positional notation, a named parameter and a later positional parameter cannot be specified for the same formal parameter.

Check that two or more named parameters cannot specify the same formal parameter (cf. IG 6.4/T3).

Check that the name used in a named parameter must only be a name of a formal parameter (cf. IG 6.4/T3).

Check that a formal parameter in a named parameter association is not confused with an actual parameter identifier having the same spelling (see IG 8.3.e/T3).

Check that a named parameter cannot be provided for a formal parameter if a positional parameter has already been given for that formal parameter (cf. IG 6.4/T3).

T82. For entries having at least one default parameter, check that (cf. IG 6.4/T4):

- calls of the form T.F(A.,B) are forbidden, where the second formal parameter has a default value;
- for a call using only positional notation, no parameters can be omitted unless the default parameters are at the end of the parameter list;
- for a call using named notation, omitted parameters must have default values;
- for a call using named notation, regardless of the order of the actual parameters, the correct correspondence with the intended formal parameter is achieved.

T83. Check that the expression corresponding to an out or an In out parameter cannot be (cf. IG 6.4.1/T1):

- a constant, including an in formal parameter, an enumeration literal, a loop parameter, a record discriminant, the literal null, and a number name;
- a parenthesized variable;
- a type conversion with a parenthesized variable;

- a function returning a value of a record, array, private, scalar, or access type;
- an attribute;
- an aggregate, even one consisting only of variables;
- a qualified expression containing only a variable name;
- an allocator;
- an expression containing an operator.

T84. For type conversions of a scalar variable as an In out parameter, check that (cf. IG 6.4.1/T3):

- **NUMERIC_ERROR** is raised for numeric types:
 - before the call when the actual value lies outside the range of the formal parameter's base type.
 - after the call when the formal's value lies outside the range of the actual variable's base type.
- **CONSTRAINT_ERROR** is raised:
 - before the call when the converted value of the actual variable lies outside the range of the formal parameter's subtype.
 - after the call when the converted value of the formal parameter lies outside the range of the actual variable's subtype.

For a type conversion of an array variable as an In out or an out parameter, check that:

- **CONSTRAINT_ERROR** is raised before the call if:
 - the subtype constraints imposed on the actual variable's components are not the same as the constraints imposed on the formal parameter's components;
Implementation Guideline: Check conversion to both a constrained and an unconstrained array type.
 - for conversion of a non-null value to an unconstrained array type, an index bound of the actual parameter, after conversion, does not lie within the range of an index subtype of the formal parameter.
 - for conversion to a constrained array type, the number of components per dimension is not the same for the formal and actual parameters when the actual variable is a non-null array, or the formal parameter specifies a non-null array.
- **NUMERIC_ERROR** is raised before the call for conversion to an unconstrained array type if the value of a bound of the variable lies outside the range of the corresponding index base type.

For a type conversion to a constrained access type as an In out parameter, check that **CONSTRAINT_ERROR** is raised:

- before the call if the value of the actual parameter is not null and the bounds or discriminants of the designated object do not equal the bounds/discriminants of the formal parameter.

Implementation Guideline: Check for both null and non-null array objects.

- after the call, if the value of the formal parameter is not null and the bounds or discriminants of the designated object do not equal the bounds/discriminants of the actual variable.

T85. For calls not involving parameters having the form of a type conversion, check that CONSTRAINT_ERROR is raised under the appropriate circumstances, namely (cf. IG 6.4.1/T4):

- before the call, when the value of a scalar in or in out actual parameter does not satisfy the range constraint of the formal parameter;
- after the call, when the value of a formal out or in out scalar parameter does not satisfy the range constraint of the actual parameter at the time of normal subprogram return;
- before the call, for all modes, when an actual record parameter has discriminant values not equal to the discriminant values of the formal parameter;

Implementation Guideline: In particular, try an unconstrained actual out parameter.

- before the call, for all modes, when an actual array parameter has different bounds for one dimension than is required by the constrained formal parameter;

Implementation Guideline: Check for null arrays with index values that are outside the index subtype.

Implementation Guideline: Check that null multi-dimensional actual parameters must have the same bounds as the formal parameter.

- for access types, when the index bounds of the object designated by the actual variable do not equal the index bounds specified for the formal parameter:
 - before the call for in and in out parameters;
 - after the call for in out and out parameters.
 - for access types, when the discriminant values of the object designated by an actual variable do not equal the discriminant values specified for the formal parameter:
 - before the call for in and in out parameters;
 - after the call for in out and out parameters.
- Implementation Guideline:* Check that within an accept statement, assignments to an out parameter obey the constraints of the formal parameter, not the constraints of the actual variable, when the constraints of the formal and the actual parameter are different.
- after the call, when the discriminant values or index bounds associated with the value of an unconstrained formal access out or in out parameter do not equal the constraint values of a constrained actual access parameter.

Check that when a private type is whose full declaration declares an access or a scalar type and is used as an out or an in out parameter, CONSTRAINT_ERROR is raised after the call if the value of the formal parameter does not belong to the subtype of the actual parameter.

Check that `CONSTRAINT_ERROR` is raised at the place of the call (i.e., within the caller, not within the called accept statement) in the above circumstances.

T86. Check that `CONSTRAINT_ERROR` is not raised under the appropriate circumstances. In particular, check that no exception is raised (cf. IG 6.4.1/T5).

- at the time of call, for all modes, when the value of a scalar actual out parameter does not satisfy the range constraints of the formal parameter;

Implementation Guideline: Check when the actual has the form of a type conversion as well as the form of a variable name.

- at the time of call, for all modes, when an actual access parameter has the value null and the formal parameter is constrained (even if the subtype of the actual parameter does not match that of the formal parameter), when the actual parameter has the form of a variable name or a type conversion;
- on normal return, for in out and out parameters, when the formal parameter value is null and the actual parameter is constrained (even if the subtypes of the formal and the actual parameters are not the same);
- for an in out or out parameter having an array type, when the formal parameter is constrained and the actual parameter has the form of a type conversion:
 - corresponding dimensions of the formal and actual parameter have the same number of components (for a non-null array), and the index bounds of the actual parameter lie outside the index subtype of the formal parameter.
 - corresponding dimensions of the formal and actual parameter do not have the same number of components, but the formal and actual parameter are both null arrays.
- for an unconstrained formal parameter having an array type when the actual parameter has a null array value and its bounds lie outside the range of the index subtype.
- for an out parameter having a constrained access type, before the call, when the object designated by the actual parameter does not satisfy the formal parameter's constraints.

T87. Check that unconstrained record, private, limited private, and array formal parameters use the constraints of the actual parameter (even when the default parameter value has a constraint different from that of the actual parameter) (cf. IG 6.4.1/T6).

Implementation Guideline: For record, private, and limited private types having default discriminant constraints, be sure to try an uninitialized constrained variable as an out actual parameter.

Implementation Guideline: Check that null strings can have negative bounds, and the bounds are passed correctly.

Check that assignments to (formal parameters of) unconstrained record types without default constraints (i.e., for which `TCONSTRAINED` is always true) raise `CONSTRAINT_ERROR` if an attempt is made to change the constraint of the actual parameter (by making a whole-record assignment to the formal parameter).

Check that assignments to (formal parameters of) unconstrained record types with default constraints (i.e., for which `TCONSTRAINED` is true or false depending on its value for the actual parameter) raises `CONSTRAINT_ERROR` if the actual parameter is constrained and the constraint values of the object being assigned do not satisfy those of the actual

parameter. Check that `CONSTRAINT_ERROR` is *not* raised if the actual parameter is unconstrained, even if the assignment changes the constraints of the actual parameter.

Implementation Guideline: Try a case where an actual parameter has the form of a type conversion.

Implementation Guideline: For both checks, include cases where the unconstrained record type is the full declaration of a private or limited private type.

Implementation Guideline: Try these checks for nested procedure calls as well, i.e., where an unconstrained formal parameter is used as an actual parameter in a subprogram call.

T88. Check that actual parameters are evaluated and identified at the time of call, e.g., use a call of the form `T.E(l, A(l))` where the parameters of `E` are out parameters, or use a name with a function prefix (cf. IG 6.4.1/T7).

T89. Check that all permitted forms of variable actual parameters are permitted (cf. IG 6.4.1/T8).

Implementation Guideline: Include a case when the variable is named by a dereferenced function, i.e., by `F.all`.

T90. Check that slices and arrays that are components of records are passed correctly to entries (cf. IG 6.4.1/T9).

Implementation Guideline: Use all parameter modes.

Implementation Guideline: Use multidimensional arrays.

Implementation Guideline: Use records with components whose bounds depend on a discriminant and have more than one array component.

Implementation Guideline: Use objects designated by access types.

Implementation Guideline: Be sure that arrays with different bounds can be passed to unconstrained formal parameters.

Implementation Guideline: Pass a formal as an actual parameter.

T91. Check that type marks appearing in actual in out or out parameters must conform to the type mark given in the formal parameter's declaration (cf. IG 6.4.1/T10).

Check that conformance is not required for in parameters (cf. IG 6.4.1/T10).

T92. Check that default values of all types (including limited types) can be passed to a formal parameter. (Nonlimited types are tested elsewhere implicitly.) (cf. IG 6.4.2/T1).

Check that `CONSTRAINT_ERROR` is raised for a default expression, if appropriate (see IG 9.5/T85).

Check that an aggregate with an `others` choice can be used as a default value for a parameter with a constrained array subtype (see IG 4.3.2/T4).

T93. Check that default expressions are evaluated each time they are needed (cf. IG 6.4.2/T2).

T94. Check that subprogram and entry redeclarations are forbidden. Use two entries and an entry and a procedure declared in the same declarative region that are identical except for one of the following differences (see IG 6.6/T1):

- the parameters are named differently (differences in parameter names are ignored).
- the subtypes of a parameter are different (differences in subtype names are ignored if the base types are the same).
- the parameter modes are different; also try reordering the parameters and changing their modes.
- a default expression is present/absent (the presence or absence of a default expression does not affect the parameter profile).

Check that an entry family name is not overloadable. In particular, check that the declaration of two entry families with different index types and different formal parameters

is not allowed if the entry families have the same identifier, nor can an entry family and single entry have the same identifier.

Check that a procedure cannot be declared in a task body with the same identifier as that of an entry family.

Implementation Guideline: Include procedures declared by a subprogram declaration, renaming declaration, and generic instantiation. Include a check when the task body is declared as a subunit.

Check that the declaration of an entry family hides outer declarations of subprograms and other entities having the same identifier.

T95. Check that overloaded subprogram/entry declarations are permitted in which there is a minimal difference between the declarations. In particular, use declarations that differ in only one of the following aspects (cf. IG 6.6/T2):

- one is a function; the other is an entry.
Implementation Guideline: Try parameterless functions and entries as well as functions and entries having at least one parameter.
- one subprogram/entry has one less parameter than the other (the omitted parameter may or may not have a default value).
- the base type of one parameter is different.
- a subprogram is declared in an outer declarative part, an entry is declared in a task, and
 - the parameters are ordered differently.
 - one subprogram/entry has one less parameter than the other, and the omitted parameter has a default value.

9.6 Delay Statements, Duration, and Time

Semantic Ramifications

S1. No upper limit is imposed on the actual duration of a delay specified in a delay statement. However, the rule in RM 9.8/4 ensures that if a high priority task executes a delay statement, it will regain control at the first opportunity after its delay has expired.

S2. The semantics of a delay statement are affected by whether it is used in a selective wait or a timed entry call (see RM 9.7). RM 9.6/1 gives the semantics of delay statements that are not part of a selective wait or a timed entry call.

S3. When TIME_OF is called with a seconds value of 86_400.0, the effect can be the same as if TIME_OF had been called with the next day and a seconds value of 0.0. To see this, consider the following statements:

```
NOW1 := TIME_OF (1984, 5, 25, 0.0);
NOW2 := TIME_OF (1984, 5, 24, 86_400.0);
```

NOW1 and NOW2 denote the same instants of time. Although the RM does not demand it, it would be reasonable for SPLIT to produce the same values for SPLIT(NOW1) and SPLIT(NOW2). In particular, most users would expect SPLIT(NOW2) to produce DAY = 25 and SECONDS = 0.0.

S4. Because DURATION is a predefined fixed point type (see Annex C), the following fixed point declaration must be allowed:

type FP is delta DURATION'SMALL range DURATION'FIRST .. DURATION'LAST;

Since values of type DURATION require at least 24 bits, this means an implementation must support fixed point values occupying at least 24 bits.

Changes from July 1982

- S5. Conditions under which TIME_ERROR is raised are stated and, in particular, expanded so "+" and "-" are expected to raise this exception under certain conditions.
- S6. The subtype, DAY_DURATION, has been declared.
- S7. The function, SECONDS, has been restricted to returning non-negative values that do not exceed 86_400.0.
- S8. The procedure SPLIT must return a SECONDS value that is in the range 0.0 .. 86_400.0.
- S9. The TIME_OF function only accepts a SECONDS value that lies in the range 0.0 .. 86_400.0.

Changes from July 1980

- S10. The maximum value of DURATION'SMALL is specified.
- S11. The subprograms SPLIT and TIME_OF have been defined.
- S12. Ordering operators are defined.
- S13. The subtypes YEAR_NUMBER, MONTH_NUMBER, DAY_NUMBER, and DAY_DURATION are defined.
- S14. Functions for determining the year, month, day, and seconds components of a TIME value are defined.

Legality Rules

- L1. The simple expression in a delay statement must have the predefined type DURATION.

Exception Conditions

- E1. TIME_ERROR is raised by (RM 9.6/6):

- the function TIME_OF if the actual parameters do not form a proper date, i.e., if the month and day are any of the following: 2/29 in a nonleap year, 2/30, 4/31, 6/31, 9/31, 11/31.
- the operator "+" if the value returned would lie outside the range January 1, 1901 through December 31, 2099.
- the operator "-" if the value returned would lie outside the range January 1, 1901 through December 31, 2099 or if the difference in times does not lie in the range of DURATION values.

- E2. CONSTRAINT_ERROR is raised by:

- TIME_OF if the value of the YEAR parameter does not lie in the range 1901 .. 2099, or the value of the MONTH parameter does not lie in the range 1 .. 12, or the value of the DAY parameter does not lie in the range 1 .. 31, or the value of the SECONDS parameter does not lie in the range 0.0 .. 86_400.0.
- "+" and "-" if called with a value of type DURATION that does not lie in the range DURATION'FIRST .. DURATION'LAST, but does lie in the range of the base type.

Test Objectives and Design Guidelines

- T1. Check that a delay statement delays execution for at least the specified time.
Implementation Guideline: Check the elapsed time by using the CLOCK function. Use small delay values, e.g., less than 90 seconds.
Check that a negative delay is accepted.
- T2. Check that the argument to the delay statement must have the type DURATION.
Check that the "+" function does not accept two arguments of type TIME.
Check that the "-" function does not allow a value of type TIME to be subtracted from a value of type DURATION.
- T3. Check that the YEAR, MONTH, and DAY parameters of TIME_OF do not have default values.
Check that the DATE parameter of SPLIT does not have a default value.
Check that the DATE parameters of YEAR, MONTH, DAY, and SECONDS do not have default values.
- T4. Check that YEAR_NUMBER, MONTH_NUMBER, DAY_NUMBER, and DAY_DURATION have the correct range constraints.
- T5. Check that addition and subtraction of values of type TIME yields the correct results when no exception is raised by the call.
Implementation Guideline: Check for values lying within a month, for values lying in different months, and for values lying in different years. The tests should take into account the fact that the range of DURATION is implementation dependent. Calls to "+" and "-" may raise CONSTRAINT_ERROR if an out-of-range DURATION value is used. A call to "-" can raise TIME_ERROR if an out-of-range DURATION value is returned.
Implementation Guideline: Check for time intervals that include the end of February in leap and nonleap years.
Check that "-" raises TIME_ERROR (not CONSTRAINT_ERROR) when the value returned does not lie in the range of DURATION'FIRST .. DURATION'LAST.
Implementation Guideline: Check both bounds of DURATION.
Check that "+" and "-" raise CONSTRAINT_ERROR when called with a value that is outside the range of DURATION'FIRST .. DURATION'LAST but within the range of DURATION'BASE'FIRST .. DURATION'BASE'LAST.
Implementation Guideline: Such a value may not exist, in which case NUMERIC_ERROR must be raised.
Check that TIME_ERROR is raised if "+" or "-" attempts to produce a value whose year number is less than 1901 or greater than 2099.
- T6. Check that comparisons are performed correctly, including equality comparisons.
Implementation Guideline: Check that no exceptions are raised even when comparing TIME values at the extremes of the permitted range.
- T7. Check that TIME_OF raises TIME_ERROR for invalid dates: 2/29 in a nonleap year, 2/30, 4/31, 6/31, 9/31, and 11/31.
Check that TIME_OF raises CONSTRAINT_ERROR if the year, month, day, or seconds arguments lie outside the specified ranges for the formal parameters.
- T8. Check that TIME_OF and SPLIT are inverse functions, except that when SECONDS has the value 86_400.0 for TIME_OF, SPLIT (and DAY) should produce a value that is one day later with zero SECONDS.
Check that the formal parameters of TIME_OF and SPLIT are named correctly.

Check that TIME_OF gives SECONDS a default value of 0.0.

Check that the functions YEAR, MONTH, DAY, and SECONDS return correct values.

Check that the formal parameter of YEAR, MONTH, DAY, and SECONDS is named DATE.

Check that the value returned by CLOCK can be processed by SPLIT.

Check that DURATION'SMALL does not exceed 0.020 seconds.

9.7 Select Statements

9.7.1 Selective Waits

Semantic Ramifications

S1. A delay statement used as a select alternative has different semantics than a delay statement used by itself. In particular, in a selective wait, a delay statement means a task is intended to wait *at most* for the specified time, whereas when used by itself, it means a task must wait *at least* the specified time.

S2. A selective wait with a zero (or negative) delay means the delay alternative is selected if there are no tasks queued at open accept alternatives. If the only open alternative is a delay alternative, then the statements of the delay alternative cannot be executed until the delay has expired.

S3. The accept alternatives need not mention different entries of the task. If two accept alternatives are open for the same entry, then one is chosen arbitrarily. Since the same entry can be mentioned several times in a single selective wait statement, it is important that the queue for an entry not be associated with accept alternatives, but instead, be associated directly with the entry.

S4. If several delay alternatives specify only slightly differing delay values, an implementation is not required to select the shortest:

```
select
    accept E;
or
    delay 1.0;
    PROC1;
or
    delay 1.0 + DURATION'SMALL;
    PROC2;
end select;
```

The condition for selecting an open delay alternative is that "the specified delay has elapsed" (RM 9.7.1/8). If SYSTEM.TICK is greater than DURATION'SMALL, it is certainly possible that both delays specified above will have elapsed between consecutive readings of the clock. In such a case, either alternative can be selected. Similarly, if a select statement is in a low priority task, then the delays specified by several alternatives might expire before the task is actually chosen for execution, at which time, any of the expired delay alternatives can be executed.

S5. If tasks with different priorities are queued on different open alternatives of a selective wait statement, RM 9.8/4 does not require that the task having the highest priority be accepted, since none of the queued tasks is "eligible for execution" until the scheduler has decided which call to

accept. The choice of entry calls is "arbitrary" (RM 9.7.1/6), and, until the choice has been made, none of the waiting tasks is eligible for execution. Of course, an implementation is allowed to take the priorities of the waiting tasks into consideration when making its "arbitrary" choice, but the RM does not require this.

S6. If a selective wait statement contains conditions, delays, and entry family indexes in accept alternatives, then the RM imposes no order of evaluation on these expressions, except that delay expressions and entry indexes must not be evaluated for closed alternatives (RM 9.7.1/5), e.g.:

```
select
  when F1 =>
    accept E(F2);
or
  when F3 =>
    delay F4;
end select;
```

The following are the permitted orders of evaluation, where F1t means that F1 is evaluated and that it returns the value TRUE, and F1f means that F1 returns the value FALSE:

```
F1t, F2, F3t, F4
F1t, F2, F3f
F1t, F3t, F2, F4
F1t, F3t, F4, F2
F1f, F3t, F4
F1f, F3f
F3t, F4, F1t, F2
F3t, F4, F1f
F3t, F1t, F4, F2
F3t, F1t, F2, F4
F3f, F1t, F2
F3f, F1f
```

S7. Since **terminate** is not a statement, it can be used only in a selective wait.

S8. Since a selective wait must contain at least one accept statement (RM 9.7.1/3), and since accept statements for an entry of a particular task cannot appear outside the task's body or inside subprograms, packages, or tasks nested in the task's body (RM 9.5/8), a selective wait statement similarly cannot appear in these places.

Changes from July 1982

S9. It is stated explicitly that if several open delay alternatives can be selected, then one is selected arbitrarily.

S10. Selective waits with a **terminate** alternative are now allowed in inner blocks that declare task objects.

Changes from July 1980

S11. An entry index or a delay alternative no longer need be evaluated immediately after the condition specified in a guard.

S12. **PROGRAM_ERROR** is raised instead of **SELECT_ERROR** when all alternatives are closed and there is no else part.

Legality Rules

- L1. At least one selective wait alternative must be an accept alternative (RM 9.7.1/3).
- L2. At most one terminate alternative is allowed in a selective wait (RM 9.7.1/3).
- L3. The innermost program unit (package, subprogram, or task) enclosing a select statement must not be a package or a subprogram, nor can it be a task that does not contain an entry or an entry family member named by an accept alternative (RM 9.7.1/3 and RM 9.5/8).
- L4. If a selective wait occurs within an accept statement, then the accept statement must not be for a single entry or an entry family that is also named by an accept alternative (RM 9.7.1/3 and RM 9.5/8).
- L5. At most one of a single terminate alternative, one or more delay alternatives, or an else part, is allowed in a selective wait statement (RM 9.7.1/3).

Exception Conditions

- E1. PROGRAM_ERROR is raised if all alternatives are closed and there is no else part (RM 9.7.1/11).

Test Objectives and Design Guidelines

- T1. Check that a selective wait statement must end with end select and not just end or end followed by an identifier.

Check that a label cannot be placed at the beginning of an accept alternative or a delay alternative.

- T2. Check that a selective wait statement must contain at least one accept alternative.
- T3. Check that the innermost unit enclosing a selective wait statement must not be a subprogram or a package (even when the selective wait names an entry of an enclosing task), or a task that does not contain an entry named by an accept alternative.
Implementation Guideline: Include a selective wait that names some entries of an enclosing task and some entries of an outer task body.

Check that within a selective wait statement, an accept alternative is not allowed to name the single entry or the entry family specified in an enclosing accept statement.

- T4. Check that terminate may not be used outside a selective wait.
- T6. Check that more than one select alternative may start with a delay statement.
- T7. Check that more than one terminate alternative is not allowed.
- T8. Check that more than one else part is not allowed.
- T9. Check that a terminate alternative and an else part are not both allowed in the same selective wait.
- T10. Check that a terminate alternative, and an alternative starting with a delay statement, are not allowed in the same selective wait.
- T11. Check that an else part and an alternative starting with a delay statement are not allowed in the same selective wait.
- T12. Check that a delay statement is allowed in the sequence of statements of a select alternative of a selective wait containing a terminate alternative or an else part.
- T13. Check that all conditions, open delay alternative expressions, and open entry family indexes are evaluated.

Implementation Guideline: Have an accept alternative without a condition. Ensure that the corresponding entry has been called, for example as follows:

```
while E'COUNT = 0 loop delay 5.0; end loop;
select
    accept E:
or when F( ) => ...
...
end select;
```

- T14. Check that for a guarded delay alternative, the delay expression is evaluated only if the guard is true.

Implementation Guideline: Report whether the delay expression is evaluated immediately after the guard is evaluated.

- T15. Check that for a guarded accept alternative for an entry family, the entry index is evaluated only if the guard is true.

Implementation Guideline: Report whether the index expression is evaluated immediately after the guard is evaluated.

- T16. Check that the conditions are not reevaluated during the wait.

Implementation Guideline: Write a select statement with a delay alternative and all other alternatives closed. Call functions in the guards and delay expression that can tell when the last function has been evaluated. When the last function is evaluated, allow another task to proceed and to change a variable whose value affects a guard.

- T17. Check that PROGRAM_ERROR is raised if all alternatives are closed and no else part is present.

Check that an else part is executed if all alternatives are closed or if there are no tasks queued for open alternatives.

Implementation Guideline: Include a case where there is a task queued for a closed alternative.

Check that an else part is not executed if a task is queued at an open alternative.

- T18. Check that a call to a closed alternative is not accepted.

- T19. Check that a terminate alternative is only selected under the correct conditions (see IG 9.4/T8).

- T20. Check that a selective wait delays at least as long as is specified in a delay alternative.

Check that if a specified delay is zero or negative and an entry call is waiting at an open alternative when the selective wait is executed, the call is accepted.

9.7.2 Conditional Entry Calls

Semantic Ramifications

S1. A conditional entry call is canceled if a rendezvous is not "immediately" possible. RM 9.7.2/4 indicates in what sense a call is immediately possible, i.e., the called task has reached an accept statement for the entry or a selective wait statement with an open accept alternative for the entry. This paragraph shows that "immediately" is not to be interpreted as "within zero seconds," but instead, is dependent only on the computational state of the called task. When the called task is executing a selective wait, more than one accept alternative might be open, and more than one task might be able to be accepted. In such a case, a conditional call need not be accepted in preference to a call waiting at a different accept alternative.

S2. In general, the calling task must wait at least until it can be determined whether the called task is both able to accept the call and has decided to accept it. In a distributed system, if the

called task is running on a different computer, the calling task might have to wait longer before knowing whether the conditional call will succeed than would be the case when the called task is running on the same computer.

S3. In short, acceptance of a conditional call is independent of the length of time required to decide if the call can be accepted; acceptance is only dependent on the state of the called task.

S4. The calling task in a conditional entry call is never entered in any queue of the called task, since entry in a queue would imply the call could not be accepted "immediately."

S5. Since the name in an entry call statement is required to be the name of an entry (i.e., a name declared by an entry declaration; see RM 9.5/4), if an entry is renamed as a procedure, the procedure name cannot be used as an entry name in a conditional entry call. This fact can be used in helping to resolve an overloading of the entry name (see IG 8.7.a.2/S).

Similarly, an entry declaration is not allowed as a generic formal declaration, and the name of a formal procedure (or function) cannot be used as an entry name in a conditional entry call even if the actual parameter is an entry name.

Changes from July 1982

S6. Evaluation of an entry name is required (rather than evaluation of just the entry index).

Changes from July 1980

S7. TASKING_ERROR is raised by a conditional entry call if the called task is completed or has been aborted.

S8. The conditions under which a rendezvous is "immediately possible" are now stated explicitly.

Legality Rules

L1. If an entry has been renamed as a procedure, the procedure name is not allowed at the place where an entry call is required in a conditional entry call (RM 8.5/9).

L2. A formal procedure parameter of a generic unit is not allowed at the place of an entry call statement in a conditional entry call (even if an actual parameter is the name of an entry) (RM 9.5/4).

Exception Conditions

E1. TASKING_ERROR is raised if the called task has already completed its execution (RM 9.7.2/6) or if the called task has been aborted (and so is abnormal) (RM 9.10/7).

Test Objectives and Design Guidelines

T1. Check that if a rendezvous requested by a conditional entry call cannot be performed "immediately," then the call is not performed. Check at least the following:

- the called task has not yet been activated;
- the queue for the called entry already contains another task;
- an accept statement for the called entry has not yet been reached.
- the body of the task containing the called entry does not contain any accept statement for that entry;
- the corresponding accept statement is permanently or temporarily closed;
- the only accept statement for the called entry appears in a selective wait with an else part (that is, the called task also refuses to wait).

Implementation Guideline: Check that the else part is executed when the call is not accepted.

- T2. Check that whenever an index is present, it is evaluated before any parameter associations are evaluated and before a rendezvous is attempted.

Implementation Guideline: Create a task without a corresponding accept statement (then the rendezvous will never be possible).

- T3. Check that a conditional entry call can appear in places where a selective wait is not allowed.

Implementation Guideline: Try procedure bodies, package bodies, and task bodies nested in tasks.

- T4. Check that TASKING_ERROR is raised if the called task has already completed its execution when the conditional entry call is executed, or if the called task is aborted prior to the call.

Implementation Guideline: In the aborted case, abort the task when a function in the entry call is called.

- T5. Check that if the rendezvous is immediately possible it is performed.

Implementation Guideline: There are two cases: when the called task is executing an accept statement, and when the called task is executing a selective wait. Use an entry family for one case.

When a call is accepted, check that the statements after the call in the conditional entry call are executed.

- T6. Check that if an entry is renamed as a procedure, the procedure name cannot be used in a conditional entry call as the name of the entry.

Check that the name of a formal procedure cannot be used in a conditional entry call as the name of the entry (see IG 12.1.3/T6).

9.7.3 Timed Entry Calls

Semantic Ramifications

S1. The RM specifies (RM 9.7.3/3) that in a timed entry call, the entry index is first evaluated, followed by the actual parameters (if any), and finally by the delay expression.

S2. There is no requirement to cancel the entry call within a minimal amount of time after the delay has expired. The RM only says that the call is canceled after expiration of the delay. Thus, if an implementation takes a certain amount of time to decide whether a timed entry call can be accepted, this amount of time can be taken even if the specified delay is zero or negative. On the other hand, an implementation might decide that if the delay is sufficiently small, no call can be accepted within that amount of time. A permissible optimization is to simply delay for that length of time. Consequently, even if the called task has reached an accept statement for the called entry, it is implementation-dependent whether the call will be allowed if the delay expression is less than some small value.

S3. In contrast, the execution of a conditional entry call must take at least long enough to determine whether the called task is able to accept the call. This means that a conditional entry call may succeed where a timed entry call with a small delay will fail. (See also IG 9.7.2/S.).

S4. Since the name in an entry call statement is required to be the name of an entry (i.e., a name declared by an entry declaration; see RM 9.5/4), if an entry is renamed as a procedure, then the procedure name cannot be used as an entry name in a conditional entry call. This fact can be used in helping to resolve an overloading of the entry name (see IG 8.7.a.2/S).

Similarly, since a generic subprogram declaration can only declare procedures or functions, the name of a formal procedure (or function) cannot be used as an entry name in a conditional entry call.

Changes from July 1982

S5. Evaluation of an entry name (not just an entry index) is now required.

Changes from July 1980

S6. `TASKING_ERROR` is raised if the called task is completed or aborted before the expiration of the delay.

Legality Rules

- L1. If an entry has been renamed as a procedure, the procedure name is not allowed at the place where an entry call is required in a conditional entry call (RM 8.5/9).
- L2. A formal procedure parameter of a generic unit is not allowed at the place of an entry call statement in a conditional entry call (even if an actual parameter is the name of an entry) (RM 9.5/4).

Exception Conditions

- E1. `TASKING_ERROR` is raised if the called task is completed at the time of the call or completes its execution before the call is accepted (RM 9.7.3/6).
- E2. `TASKING_ERROR` is raised if the called task is aborted before the call is accepted (or while performing the rendezvous) (RM 9.10/7).

Test Objectives and Design Guidelines

- T1. Check that a timed entry call delays for at least the specified amount of time if a rendezvous is not possible. Check at least the following conditions:

- the called task has not yet been activated;
- the queue for the called entry already contains another task whose rendezvous cannot be completed within the specified delay (e.g., because the rendezvous contains a delay statement);
- an accept statement for the called entry has not yet been reached;
- the body of the task containing the called entry does not contain any accept statement for that entry;
- the corresponding accept statement is permanently or temporarily closed;
- the only accept statement for the called entry appears in a selective wait with an else part (that is, the called task refuses to wait).

Implementation Guideline: Check that the delay is at least for the specified time by calling `CLOCK` before and after the timed entry call.

- T2. Check that whenever an index is present it is evaluated before any parameter associations are evaluated, and the parameter associations are evaluated before the delay expression. Then a rendezvous is attempted.

Implementation Guideline: Create a task without a corresponding accept statement (then the rendezvous will never be possible).

- T3. Check that a timed entry call can appear in places where a selective wait is not allowed.

Implementation Guideline: Try procedure bodies, package bodies, and task bodies nested in tasks.

- T4. Check that `TASKING_ERROR` is raised if the called task has already completed its execution when the timed entry call is executed, or if the called task is aborted prior to the call.

Implementation Guideline: In the aborted case, abort the task when a function in the entry call is called.

- T5. Check that if the rendezvous is immediately possible (and the delay is lengthy, e.g., more than 10 seconds), then the rendezvous is performed.

Implementation Guideline: There are two cases: when the called task is executing an accept statement, and when the called task is executing a selective wait. Use an entry family for one case.

When a call is accepted, check that the statements after the call in the entry call are executed.

Check also, that if a call is not immediately possible, but becomes possible before the delay expires, the call is accepted.

- T6. Check that if an entry is renamed as a procedure, the procedure name cannot be used in a timed entry call as the name of the entry.

Check that the name of a formal procedure cannot be used in a conditional entry call as the name of the entry (see IG 12.1.3/T6).

- T7. Check that a canceled entry call is removed from the queue of the called task's entry.

Implementation Guideline: Create a queue with a normal call, two timed calls, and a normal call, and check that the appropriate calls are accepted if one or both of the timed calls are canceled.

9.8 Priorities

Semantic Ramifications

S1. Since RM 2.8/9 states that a pragma "has no effect ... if its placement or its arguments do not correspond to what is allowed for the pragma," the pragma PRIORITY is ignored if it is used in an inappropriate place or has a nonstatic argument. The program is not illegal. A friendly compiler will warn a user when a pragma is ignored.

S2. If the value specified in a pragma PRIORITY lies outside the range of permitted priority values, no exception is raised, since such a value does "not correspond to what is allowed for the pragma." Similarly, if a nonstatic expression is used, it is not evaluated.

S3. Although the RM requires that the argument to the pragma PRIORITY be a static integer expression, the RM leaves completely open how an implementation chooses to define the subtype PRIORITY. In particular, the subtype could be defined as a null range, so no value would be recognized as a suitable priority value. An implementation is even free to define PRIORITY as a nonstatic subtype, although this might make it difficult to decide when the pragma should be ignored.

S4. Even if no user-defined priorities are supported (by making subtype PRIORITY a null range), it is still required that entry calls associated with interrupts be treated with a higher priority than other calls (RM 13.5.1/2).

S5. If tasks with different priorities are queued on different open alternatives of a selective wait statement, RM 9.8/4 does not require that the task having the highest priority be accepted, since neither task is "eligible for execution" until the scheduler has decided which call to accept. The choice of entry calls is "arbitrary" (RM 9.7.1/6), and until the choice has been made, none of the waiting tasks is eligible for execution. Of course, an implementation is allowed to take the priorities of the waiting tasks into consideration when making its "arbitrary" choice, but the RM does not require this.

S6. If tasks of different priorities are queued for the same entry, their requests for rendezvous are serviced in the order of arrival (RM 9.5/15); higher priority tasks are not serviced first unless they arrived first.

S7. It is possible for a high priority task to be blocked by the activation of a lower priority task. For example:

```

task type LOW is
  pragma PRIORITY (PRIORITY'FIRST);  -- least urgent
end LOW;

type A_LOW is access LOW;

task body LOW is
  ...      -- activation actions here
begin ... end LOW;

task MED is
  pragma PRIORITY (PRIORITY'FIRST+1);
end MED;

task body MED ... end MED;

task HIGH is
  pragma PRIORITY (PRIORITY'LAST);  -- most urgent
end HIGH;

task body HIGH is
  X_LOW : A_LOW := new LOW;          -- (1)
  ...                                -- (2)
begin ... end HIGH;

```

Tasks MED and HIGH are to be activated. Since HIGH has the highest priority, its activation starts first. Among its first activation actions is the activation of the low priority task at (1) via an allocator. The activation of HIGH cannot proceed until LOW is activated (RM 9.3/6). But since task MED is also awaiting activation, and since MED has a priority higher than that of LOW, RM 9.8/4 requires that the activation of LOW be suspended (and hence, the activation of HIGH be delayed) in favor of beginning the activation of MED. Assuming that MED does not require the activation of any tasks before completing its own activation, the activation of LOW will only resume when MED is fully activated. In short, since tasks are not activated with the priority of their master, it is possible for a high priority task's activation to be delayed by the need to activate a lower priority task.

S8. RM 9.8/6 says priorities should not be used for task synchronization. This means that programmers should not assume that tasks with different priorities each have exclusive access to a shared variable because only one task can be running. Although this reasoning may be correct for a particular Ada implementation, it is not correct for all implementations, since priorities have no effect if each task executes on its own CPU.

S9. Priorities have no detectable semantic effect when there is no competition for resources, i.e., when the number of tasks eligible to run never exceeds the available resources. Consequently, it is not possible to write implementation-independent tests to check whether priorities are being obeyed.

Changes from July 1982

S10. The pragma PRIORITY is allowed in any library unit subprogram and is ignored if this subprogram is not used as a main program.

Changes from July 1980

S11. There are no significant changes.

Test Objectives and Design Guidelines

T1. Check that if the expression in a pragma PRIORITY is static, and is outside the range of the subtype PRIORITY, no exception is raised.

Check that if the expression in a pragma PRIORITY is nonstatic, it is not evaluated (since the pragma has no effect).

T2. When calls are queued on several open alternatives of a select statement, check to see if priorities seem to be considered in deciding which call to accept.

T3. Check to see if a rendezvous is executed with the priority of the calling task instead of with the priority of the called task.

Implementation Guideline: If this check appears to indicate that the rendezvous is not executed with the correct priority, then an implementation's scheduling algorithm should be examined more carefully to see if it conforms to RM 9.8/4. Any test written for this objective may report failure if there is, in fact, no contention for resources, since in this case, priorities have no effect.

9.9 Task and Entry Attributes

Semantic Ramifications

S1. The prefix of the 'CALLABLE and 'TERMINATED attributes may be a task object (i.e., an object declared by an object declaration or the declaration of a formal parameter of a subprogram, entry, or generic unit, or a subcomponent of such an object) or it may be a value (i.e., a value returned by a function call, or subcomponent of such a value). The prefix may also have an access type if the designated type is a task type (RM 3.8.2/3).

S2. The definition of 'COUNT in RM 9.9/6 says simply that it returns "the number of entry calls presently queued on the entry [denoted by the prefix]." Now suppose entry family E has one task waiting on E(1) and one waiting on E(0). Consider the following uses of the 'COUNT attribute:

```
accept E(E(1)'COUNT) do      -- (1) which call is accepted?
  if E(1)'COUNT = 1 then    -- (2) is the condition TRUE?
```

When the first accept statement is evaluated, E(1)'COUNT equals 1, since there is one call waiting, so the call for entry E(1) is accepted. Once this call is accepted, there is no longer a call waiting in the queue, so E(1)'COUNT now equals zero. Consider the parenthesized comment in RM 9.9/6: "(if the [COUNT] attribute is evaluated by the execution of an accept statement for the entry, then the count does not include the calling task)." This comment applies to the use of the 'COUNT attribute at point (2), since the syntax of an accept statement includes the sequence of statements following do, and since the calling task is no longer in the queue. It is not a correct statement of the rule for the use at (1), since when the attribute is evaluated at (1), no calling task has been removed from any queue — the evaluation simply serves to determine which queue should be considered.

S3. If an entry is renamed as a procedure, then the new name cannot be used as the prefix of the 'COUNT attribute, since it is considered the name of a procedure (RM 8.5/9). On the other hand, if a task is renamed, the new name can be used as a prefix for the 'CALLABLE and 'TERMINATED attributes.

S4. The prefix of the 'COUNT attribute must be the name of an entry in an enclosing task unit. It cannot be the name of an entry of a task object:

```

task type T is
    entry E;
end T;

```

```

OBJ : T;

```

```

task body T is
    CNT : INTEGER;
begin
    CNT := OBJ.E'COUNT;    -- illegal prefix
    CNT := T.E'COUNT;      -- legal prefix
    CNT := E'COUNT;        -- legal prefix
end T;

```

OBJ.E is an illegal prefix because OBJ is not the name of an enclosing task unit; it is the name of an object. On the other hand, since T is the name of the enclosing task unit, T.E is just an expanded name for an entry of the enclosing unit, and so this prefix is okay.

S5. If X is a task object, then X'SIZE returns the size needed to hold a value that designates a task, i.e., it does not return the size of the designated task. If X is declared as a single task, i.e., with the declaration task X, then X denotes an implicitly declared task object (see RM 9.1/2), and X'SIZE returns the size of this implicitly declared object.

S6. If T is the name of a task type, then T'SIZE returns the minimum number of bits needed to hold any value that designates a task having type T.

S7. Within the body of a task type, the task's identifier denotes a task object (RM 9.1/4), and so T'SIZE is allowed and returns the size of the task object whose designated task is currently executing the task body.

S8. If X is a task object, X'ADDRESS returns the address of the object designating a task, not the address of the task's machine code. If X is declared as a single task, then X'ADDRESS returns the address of the implicitly declared object associated with the task's declaration.

S9. Outside the body of a task type, T, the task's identifier denotes a task type as well as a task unit. Since T'ADDRESS is not legal for a task type, it returns the address of the machine code for T's body. Within the body of T, T is used as both the name of the object executing the body and as the name of the enclosing task unit. Within the body, T'ADDRESS should be interpreted as the address of the enclosing unit.

S10. The prefix of 'SIZE cannot be either the name of a function or a function call. A function call is not allowed because such a call produces a value, not an object. A function name (i.e., a prefix having the form of a parameterless function call) is not allowed since it cannot be invoked (it would produce a value) and since the name itself is not the name of an object.

S11. If a prefix, P, is an object of an access type whose designated type is a task type, then P'SORAGE_SIZE yields the collection size for P's type, not the number of storage units reserved for the activation of the designated task object. P.all'SORAGE_SIZE, however, does return the number of storage units reserved for the designated task's activation.

S12. Prior to a task's activation, T'CALLABLE is TRUE and T'TERMINATED is FALSE:

```

task T;
B1 : BOOLEAN := T'CALLABLE;    -- yields TRUE
B2 : BOOLEAN := T'TERMINATED;  -- yields FALSE

```

Changes from July 1982

S13. The 'COUNT attribute can be used within an accept statement for the entry named in its prefix.

Changes from July 1980

S14. 'CALLABLE has been added.

S15. The prefix of the 'TERMINATED attribute cannot be a task type.

S16. The attribute 'PRIORITY no longer exists.

S17. For entry E of task unit T, the 'COUNT attribute is now not allowed within subprograms, packages, or tasks nested within T.

S18. 'COUNT now returns a value of type *universal_integer*.

S19. The prefix of the 'COUNT attribute must be the name of a task unit, not the name of a task object or value.

Legality Rules

- L1. The prefix of the 'CALLABLE and 'TERMINATED attributes must have a task type (RM 9.9/1) or an access type designating a task type (RM 3.8.2/3).
- L2. The prefix of the 'COUNT attribute must be the name of an entry (either a single entry or a member of an entry family) of task unit T, and the attribute must appear within the body of T, but not within any subprogram, package, or task nested within T (RM 9.9/5).
- L3. The prefix of the 'COUNT attribute cannot be a name declared by a renaming declaration (RM 8.5/9).

Exception Conditions

- E1. *CONSTRAINT_ERROR* is raised if the prefix of CALLABLE or TERMINATED has the value null (RM 4.1/10).

Test Objectives and Design Guidelines

- T1. Check that the prefix of the attributes 'TERMINATED and 'CALLABLE cannot be the name of a task type.
Implementation Guideline: Include a check that the prefix can be the name of a task unit inside the body of a task type.
- T2. Check that the prefix of the 'COUNT attribute must be the name of an entry of the innermost task unit enclosing the use of the attribute.

Check that the prefix cannot be the name of a task object, even if the object has the type of the enclosing task unit.

Check that 'COUNT cannot be used in an inner procedure, package, or task.
- T3. Check that when the prefix of the 'COUNT attribute contains the name of an entry family, the entry name must have the form of an indexed component.
- T4. Check that the prefix of 'TERMINATED and 'CALLABLE can be a function call returning either a value having a task type or an access value designating a task.

Check that the prefix can be an object having a task type or having an access type that designates a task.

Check the values returned by 'TERMINATED and 'CALLABLE (checked implicitly by tests in IG 9.10/T).

- T5. Check that 'COUNT returns the correct value, both for single entries and for members of an entry family.

Implementation Guideline: Include a check when 'COUNT occurs as an entry family index in an accept statement.

- T6. Check that 'COUNT returns a value of type *universal_integer*.

- T7. Check that the prefix of 'STORAGE_SIZE can be a task type or a task object (including a formal parameter).

Implementation Guideline: Include a check that the name of a task type or subtype can be used as a prefix within the task's body. Also use the name of a single task.

Check that an object having an access type is not allowed as the prefix of 'STORAGE_SIZE, even if the designated object has a task type, nor can the prefix be a function call or the name of a function (even if the function returns a task value).

Check that the prefix of 'SIZE can be a task object or a task type.

Implementation Guideline: Include a check that the name of a task type or subtype can be used as a prefix within the task's body. Also use the name of a single task.

Check that the prefix of 'ADDRESS can be a task object (a single task, an object having a user-defined task type, or a formal parameter) or the name of a task unit (only applies outside the task body to names of task types).

Implementation Guideline: Check that task objects have different addresses. Also, check that when two objects having the same task type are called, 'ADDRESS returns the same values within the task body when the name of the task unit is used.

Check that the prefix of 'ADDRESS cannot be a subtype name for a task type, whether the name is used within or outside the task body.

- T8. Check that CONSTRAINT_ERROR is raised if the prefix of CALLABLE or TERMINATED has the value null (see IG 4.1.4/T1).

9.10 Abort Statements

Semantic Ramifications

S1. After an abort statement has completed its execution, the aborted task may continue to execute (in particular, it may read and write shared variables) but it cannot activate any new tasks or call any other task. The aborted task must be removed from any queues before completion of the abort statement, so no entries can be called by an aborted task (RM 9.10/5).

S2. The reason an aborted task is considered abnormal rather than complete is to allow the execution of the aborting task to continue before the aborted task is actually terminated. For example, if a task running on a remote node in a distributed system is aborted, then the node executing the abort statement can send a message to the remote node specifying which task is to be aborted (after removing it from a queue, if necessary), and then, without waiting to be sure that the task has actually been terminated, it can continue — after noting in its own tables that the aborted task is no longer callable. The aborted task can continue to execute until it reaches a synchronization point (RM 9.10/6).

S3. Since the activation of a task is a synchronization point, an aborted task can never activate additional tasks before it completes.

S4. If an aborted task is in a loop and never reaches a synchronization point, the abort need never take effect, in the sense that the task can continue to execute. However, calls to such a task will raise TASKING_ERROR rather than be placed in a queue.

S5. If a calling task is aborted while a rendezvous is in progress, the rendezvous is allowed to

complete (RM 9.10/6). This ensures that the called task has access to actual parameters while the rendezvous is in progress, even though the caller will disappear after the rendezvous is completed. Since the rendezvous is not terminated prematurely, if the rendezvous is in a loop or is executing a long delay, it can take some time for the calling task to become terminated. Of course, a programmer can always abort the called task as well, to allow the calling task to terminate.

S6. If a task is aborted while updating a global variable, the task need not complete the update before the task terminates (RM 9.10/8). This means that updating need not be implemented as an indivisible operation. Moreover, since updating or reading a variable named in a SHARED pragma is not one of the synchronization points listed in RM 9.10/6, such updates or reads can occur even after a task has become abnormal.

S7. If a task name in an abort statement is a function call, or if the name includes a function call, e.g., F.T, then overloads of the function name can be resolved using the fact that the name must denote a task.

S8. A task can be aborted before it is activated (see IG 9.3/S).

Changes from July 1982

S9. Aborted tasks become complete rather than terminated (so the usual rules for task termination can be used).

S10. The raising of TASKING_ERROR exceptions in callers of aborted tasks is no longer part of the execution of the abort statement, but instead occurs independently, and so can be raised before the aborted task completes its execution.

S11. The activation of a task is now considered a synchronization point.

Changes from July 1980

S12. Aborted tasks become abnormal rather than being terminated directly.

Legality Rules

L1. Each name in an abort statement must have a task type (RM 9.10/3).

Exception Conditions

E1. TASKING_ERROR is raised in the calling task if an entry of an abnormal task is called (RM 9.10/7).

E2. TASKING_ERROR is raised in the calling task if the called task becomes abnormal before the call has been accepted or while executing the rendezvous (RM 9.10/7).

Test Objectives and Design Guidelines

T1. Check that at least one task name is required in an abort statement.

T3. Check that aborting a terminated task does not cause an exception.

T4. Check that if a task is aborted before being activated, the task is terminated, and no exceptions are raised.

T5. Check that after completing execution of the abort statement, the named tasks and all dependent tasks are not callable, i.e., 'CALLABLE is FALSE.

T6. Check that a task may abort itself; check that the next statement is not executed.

Implementation Guideline: The abort must be in a conditional statement, otherwise the compiler may delete all following statements.

- T7. Check that a task may abort a task that it depends on.
- T8. Check that an aborted task is removed from any entry queue it may be on.
Implementation Guideline: Check that the queue is maintained correctly when the aborted task is at the front of the queue, in the middle, or at the end.
- T9. Check that if a calling task is aborted during a rendezvous, the rendezvous must be completed. In particular, check that if the master of a calling task is aborted while a dependent task is in a rendezvous, neither the master nor the dependent task can be terminated while the rendezvous continues.
- T10. Check that an abnormal task can be aborted.
- T11. Check that TASKING_ERROR is raised in the calling task if a called task is aborted while in a rendezvous.

Check that TASKING_ERROR is raised by a timed entry call if the called task is aborted before the delay expires, but not when the call is first executed.

Check that TASKING_ERROR is raised by a conditional entry call if the called task is abnormal at the time of the call (see IG 9.7.2/T4).

9.11 Shared Variables

Semantic Ramifications

S1. The significance of the assumptions stated in RM 9.11/3-6 is that when a shared variable is a scalar or an access type, a compiler can generate code to read and write the variable on the assumption that a particular task has exclusive access to the variable between synchronization points. In particular, if a task has exclusive access to a variable, it can assume that the value of the variable does not change unless it has been assigned to. A compiler can therefore keep local copies of a variable's value in registers and can defer reading or updating the actual variable until the register is needed for some other purpose, or until a synchronization point is reached. At each synchronization point, any updates to shared variables must be made, and it is possible that the value of a shared variable will be changed by some other task (so its value must be read again if it is needed).

S2. The pragma SHARED makes each read or update of a shared variable a synchronization point for that variable, i.e., every read must actually read the value of the variable (which may have been changed by some other task since the last read), and every update must be stored home immediately in the variable rather than being retained in a register.

S3. The current formulation of the assumptions in RM 9.11/3-6 is defective with respect to variables that are subcomponents of a composite type. For example:

```
type BITS is array (0..15) of BOOLEAN;
pragma PACKED (BITS);
WORD : BITS;
```

```
task T1;
task T2;
```

```
task body T1 is
begin
  WORD(14) := TRUE;
end T1;
```



```

task body T2 is
begin
  WORD(15) := FALSE;
end T2;

```

According to RM 9.11/3-6, the assignments to the different components of WORD are not erroneous because WORD(14) and WORD(15) are different variables. Of course, in most implementations, the packed BITS array will occupy exactly 16 bits, and for a byte addressable machine, WORD(14) and WORD(15) will occupy the same byte. The only way to modify one bit in a byte is to read the byte, modify the bit, and store back an entire byte. In effect, the assignment to WORD(14), say, would usually be implemented as:

```
WORD(8..15) := SET_BIT_14_OF (WORD(8..15));
```

Each task would, in effect, be assigning to the same slice of the WORD array.

S4. The above implementation of assignments to components of WORD is incorrect because incorrect results can be obtained, i.e., WORD(14) might not have the value TRUE after the assignment. An implementation has two choices here. It can refuse to obey the PACKED pragma, and store BITS arrays so each component occupies one addressable storage unit, or it can establish a semaphore guarding access to the shared variable WORD, to ensure that updates of individual components are made indivisibly. Note that the same issue arises for record types having an explicitly defined representation that requires two record components to occupy the same addressable storage unit.

S5. The Ada Language Maintenance Committee has considered this situation and has recommended that paragraphs 3-6 be understood to include variables whose representation has been specified with an explicit representation clause or PACKED pragma. This means that the assignments to WORD(14) and WORD(15) above would be considered erroneous, i.e., it is up to the programmer to ensure that they are performed on a noninterfering basis. An implementation is allowed to use the straightforward implementation that was originally suggested.

S6. The name given in a pragma SHARED cannot be a name declared by a renaming declaration even if the renamed entity is an object declared by an object declaration (and is a variable having a scalar or an access type). The wording does not permit such names since the rule in RM 9.11/10 only allows names declared by object declarations.

S7. Although a name declared by a renaming declaration cannot be used in a pragma SHARED, the effect of a pragma SHARED extends to names declared by renaming declarations, i.e., if a renamed entity has been named in a pragma SHARED, reads and updates using the new name are also considered synchronization points for the new name. The reason is that the pragma SHARED specifies that every read or update "of a variable" is a synchronization point. A variable is an object (RM 3.2.1/3) and an object is an entity (RM 3.1/1). A renaming declaration declares a new name for an entity (RM 8.5/1). So the pragma SHARED applies to the entity named in the pragma, and consequently, applies to any new names for the entity as well.

S8. Since the term "shared variable" refers to variables accessible from more than one task (RM 9.11/2), it is convenient to have a different term for those variables named in a pragma SHARED. We shall call such variables "volatile variables." The term refers to the fact that from an implementation's point of view, the value of such a variable can change without being assigned to.

S9. If a volatile variable is used as an actual generic in out parameter, then all reads and updates of the formal parameter within the instantiated unit are considered synchronization points for the variable. In general, this means that the code generated for such an instantiation

will be different from the code generated for an instantiation that does not refer to a volatile variable.

S10. If a subprogram or an entry is called with a volatile in out or out actual parameter, then for the in out parameter, a read synchronization point occurs when the variable is passed to the formal parameter, and an update synchronization point occurs when the subprogram returns. In particular, the formal parameter is a different object (since volatile variables must have scalar or access types and values of these types are passed by copy), so each read or update of the formal parameter is not a synchronization point. Code generated within the subprogram or rendezvous is not affected by the fact that an actual parameter may be a volatile variable.

S11. The synchronization points introduced by the use of the pragma SHARED are not considered when a task is aborted, since these points are not listed in RM 9.10/5-6. This means that an aborted task may update a shared variable if such an update occurs after the task has been aborted and before the task has reached one of the synchronization points mentioned in RM 9.10/6.

S12. If the arguments to a pragma do not correspond to what is expected or allowed for a pragma, the pragma then is ignored; the program is not illegal (RM 2.8/9), e.g.:

```
pragma SHARED (A, B, C);      -- ignored; not illegal
pragma SHARED (A(3));        -- ignored; not illegal
pragma SHARED (A, B, );      -- syntactically illegal
```

The last example does not conform to the syntax for a pragma as specified in RM 2.8/2, and is illegal because the program containing it does not conform to the syntax of a legal Ada program.

Changes from July 1982

S13. Volatile variables are indicated with a pragma rather than by instantiating a generic procedure. In addition, such variables cannot be formal parameters of subprograms or generic units, objects designated by an access value, or subcomponents of composite objects.

S14. Volatile variables must only have scalar or access types.

S15. Indivisible access must be provided for volatile variables (or the pragma is ignored).

Changes from July 1980

S16. The wording makes clearer that violation of permitted assumptions about shared variables causes a program to be considered erroneous.

Legality Rules

Violation of these rules means that the pragma is ignored; see RM 2.8/9.

- L1. The variable named in a pragma SHARED must have a scalar or an access type and must be declared by an object declaration (RM 9.11/10).
- L2. The pragma SHARED must occur immediately within the declarative part or package specification in which the named variable is declared (RM 9.11/10).
- L3. The pragma SHARED must occur before any occurrence of the name of the variable, other than in an address clause.

Test Objectives and Design Guidelines

- T1. Check that more than one name is not allowed in a SHARED pragma.
- T2. Check that the SHARED pragma is ignored (and the name is not evaluated) if the name denotes a variable designated by an access value or denoted by a formal parameter of a subprogram or generic unit, or a subcomponent of such a variable.

Check that the pragma is ignored if the name in the pragma is declared by a renaming declaration.

Chapter 10

Program Structure and Compilation Issues

10.1 Compilation Units -- Library Units

Semantic Ramifications

S1. RM 10.1/6 says "a subprogram body given in a compilation unit is interpreted as a secondary unit if the program library already contains a library unit that is a subprogram [or a generic subprogram] with the same name." Otherwise, the subprogram body is interpreted as the declaration of a library unit together with the corresponding library unit body. (The interpolated phrase is needed to show that the rule applies to generic subprograms as well as to nongeneric subprograms; if the phrase is not considered present, it would be impossible to separately compile a body for a generic library subprogram. Each attempt to compile a body would instead declare a new nongeneric subprogram.)

S2. There are several important implications of RM 10.1/6. First, the initial compilation of a subprogram body serves to declare the subprogram as a library unit. Subsequent compilations of subprograms with the same name must conform to the declaration implied by the first compilation:

```

procedure P (X : INTEGER) is
begin ... end P;           -- initial compilation

procedure P (Y : FLOAT) is   -- illegal; nonconforming
begin ... end P;

```

The second compilation unit must be rejected because it does not conform with the specification that was implicitly declared when P was first compiled. Note that an attempt to compile a function body named P will similarly be rejected. Such attempted compilations are not considered as declaring new subprograms named P because P already exists in the library.

S3. Having compiled subprogram P, a generic unit or a package named P can subsequently be compiled without error; such a compilation removes the subprogram declaration and body from the library and redefines P to refer to the new compilation unit (RM 10.1/4).

S4. If a library already contains a package or a generic package named P, then compilation of a subprogram body P is allowed and serves to declare a new library unit named P;

```

package P is
  ...
end P;

function P return INTEGER is -- legal; replaces package P
begin ... end P;

```

S5. If a library already contains a generic subprogram named P, then any subsequent compilation of a subprogram body with the name P is considered a secondary unit for the generic subprogram. Consequently, such subprogram bodies must conform to the specification given for the generic unit:

```

generic
procedure P (X : INTEGER);

```

```

procedure P (Y : FLOAT) is      -- illegal; nonconforming
begin ... end P;

```

S6. Now suppose the subprogram body contains a context clause:

```

package P is
  subtype T is INTEGER;
end P;

with P;
function F (X : P.T) return INTEGER is
begin ... end F;

```

The compilation unit for F generates an implicit declaration of F that is equivalent to:

```

with P;
function F (X : P.T) return INTEGER;

```

Suppose P.T is modified (e.g., changed from **INTEGER** to **STRING**) and we recompile P. The recompilation of P makes F's declaration obsolete (RM 10.3/5):

"A compilation unit is potentially affected by a change in any library unit named by its context clause. ... Compilation units potentially affected by [the successful recompilation of another unit] are obsolete and must be recompiled unless they are no longer needed."

An obsolete unit must not be used and therefore acts as if it were removed from the library. Consequently, an attempt to recompile F's body (including the with clause) must succeed, because there is no longer a subprogram named F in the library. The recompilation will declare a new F that uses the new definition of P.T. An attempt to compile F's body without a with clause naming P will fail since the P in P.T is not known as the name of a package after F's declaration is made obsolete.

S7. The requirement that library units have unique names (RM 10.1/3) implies that subprogram library units cannot be overloaded; recompilation of a subprogram declaration with a different set of parameters simply replaces the earlier definition (including its subunits, if any; see RM 10.3/5).

```

procedure P (X : INTEGER);
procedure P (X : FLOAT);      -- replace first declaration
procedure P (X : INTEGER) is -- illegal; nonconforming
begin ... end P;

```

The procedure body is illegal because the only library unit declaration of P that exists has a parameter of type **FLOAT**.

S8. The name of a library unit or a secondary unit must be an identifier since such units must denote subprograms, packages, generic units, or task bodies, and RM 10.1/3 requires an identifier for a library unit subprogram; RM 12.1/4 requires an identifier for a generic subprogram; RM 7.1/2 requires an identifier for a package (including generic packages); and RM 9.1/3 requires that tasks be named with identifiers.

S9. Certain library units are predefined in the language (see Annex C) — **STANDARD**, **CALENDAR**, **SYSTEM**, **MACHINE_CODE** (if provided by an implementation), **UNCHECKED_DEALLOCATION**, **UNCHECKED_CONVERSION**, **SEQUENTIAL_IO**, **DIRECT_IO**, **TEXT_IO**, **IO_EXCEPTIONS**, and **LOW_LEVEL_IO**. Since there is no rule forbidding recompilation of these units, it is permitted, even though such a recompilation would make the predefined units inaccessible. The recompilation of **SYSTEM**, in addition, may make some predefined library units obsolete, e.g., **TEXT_IO**'s body might depend on the **SYSTEM** package. Recompilation of **SYSTEM** would then make **TEXT_IO** unusable.

S10. Library units, as well as units mentioned in with clauses, are implicitly declared in STANDARD (see RM 8.6/2), so no library unit can have the same name as any identifier declared in STANDARD (since this would introduce a forbidden redeclaration in STANDARD); however, the identifiers TRUE and FALSE can be declared as library subprograms as long as they are not declared as parameterless functions returning predefined type BOOLEAN. (Note: predefined TRUE and FALSE are not library units, although they are subprograms implicitly declared in STANDARD. They are not library units because only certain compilation units are library units (RM 10.1/3). Since TRUE and FALSE were not declared as compilation units, they are not library units; with TRUE, for example, would be illegal in the absence of a user-defined library unit named TRUE.)

S11. Since a compilation unit can begin with more than one with clause, a compiler can provide an optional compilation mode in which access to certain library units is implicitly provided by (in effect) inserting a with clause (and optionally, a use clause), naming these units in front of every unit compiled. These implicit insertions could provide automatic access to a standard set of library units, e.g., the TEXT_IO package, a package of mathematical functions, or an application-dependent set of library packages. Such a capability would have the same effect as invoking a preprocessor that modifies compilation units before they are compiled. Nothing in the language specification forbids or requires such a "preprocessor" option, but it is one of the reasons that more than one with clause is permitted at the beginning of compilation units. Of course, the use of such "options" must be under user control or the implementation will be nonstandard.

Changes from July 1982

S12. A subprogram body submitted as a compilation unit is no longer interpreted as a secondary unit if any name P has been previously declared as a library unit; it is a secondary unit only if the previous declaration of P is a subprogram or a generic subprogram.

Changes from July 1980

S13. The definition of a main program is clarified.

Legality Rules

- L1. A subprogram that is a library unit or a subunit cannot have an operator_symbol as its designator.
- L2. No library unit can have a name identical to one of the following names: BOOLEAN, INTEGER, FLOAT, CHARACTER, ASCII, NATURAL, POSITIVE, STRING, DURATION, CONSTRAINT_ERROR, NUMERIC_ERROR, PROGRAM_ERROR, STORAGE_ERROR, or TASKING_ERROR (see IG 8.6/T1).
- L3. No library unit can be a parameterless function named TRUE or FALSE that returns values having the type predefined BOOLEAN.
- L4. A package body must not be compiled before its specification has been compiled.
- L5. A library unit package cannot be executed as a main program.

Test Objectives and Design Guidelines

- T1. Check that
 - a. a subprogram cannot be compiled as a library unit or as a subunit if its designator is an operator_symbol.
 - b. no library unit can be named BOOLEAN, INTEGER, FLOAT, CHARACTER, ASCII, NATURAL, POSITIVE, STRING, DURATION, CONSTRAINT_ERROR,

NUMERIC_ERROR, PROGRAM_ERROR, STORAGE_ERROR, or
TASKING_ERROR.

- c. a library package body cannot be compiled before its specification has been compiled.
- d. an implicit use clause (preceded by a with clause) is not provided for any of the standard packages, namely, CALENDAR, TEXT_IO, IO_EXCEPTIONS, SYSTEM, MACHINE_CODE, and LOW_LEVEL_IO.
- e. an implicit with clause is not provided for any of the predefined units CALENDAR, MACHINE_CODE, SYSTEM, UNCHECKED_DEALLOCATION, UNCHECKED_CONVERSION, SEQUENTIAL_IO, DIRECT_IO, TEXT_IO, IO_EXCEPTIONS, and LOW_LEVEL_IO.
- f. overloaded subprograms are not allowed as library units (nor as subunits; see IG 10.2/T1 for subunit test).
- g. task specifications and bodies cannot be separately compiled.
- h. a package cannot be named as a main program.
- i. a package body cannot be compiled if its specification has not been first compiled and placed in the library (see IG 10.3/T1).

- T2. Check that a subunit can have the same name as a library unit or an identifier declared in STANDARD.

Check that a subprogram named TRUE or FALSE can be declared as a library subprogram if it is not a parameterless function returning values of type BOOLEAN (see IG 8.6/T1).

- T3. Check that more than one completely independent compilation unit can be submitted to a compiler in a single file. Check whether the presence of an illegal unit in a compilation affects the compilation of another, independent unit submitted in the same compilation.
- T4. Check that a package specification and body can be submitted together for compilation.
- T5. Check that a subprogram specification and body can be submitted together for compilation.
- T6. Check that a library unit and its subunits can be submitted together for compilation.
- T7. Check that a nongeneric package specification can be compiled without its body.
- T8. Check that a generic package specification can be compiled and instantiated without its body.
- T9. Check that a nongeneric subprogram declaration can be compiled without its body.
- T10. Check that the specification of a separately compiled subprogram body must conform to that of the declared specification.

Implementation Guideline: Check for generic and nongeneric units.

- T11. Check that if a subprogram body is initially compiled, subsequent attempts to compile a subprogram body with a different parameter and a result type profile are allowed (AI-00199).

Implementation Guideline: See IG 6.6/T2 for minimally different parameter and result type profiles.

Check that after compiling a generic or nongeneric subprogram specification, subsequent attempts to compile a subprogram body with a different parameter and result type profile are rejected.

T12. Check whether a generic subprogram body can be submitted separately from its declaration for compilation.

T13. Check that a generic package or subprogram instantiation can be submitted for compilation.

Implementation Guideline: Note that pragma ELABORATE will need to name the generic library unit to ensure that the unit's body is elaborated before the instantiation is elaborated.

T14. Check that a subunit can be submitted for compilation in a separate file.

T19. Check that in a compilation containing several compilation units, the order of elaboration need not be the same as the order of compilation (see IG 10.5/T2).

T20. Check that if a subprogram P is compiled, subsequent compilation of a generic package, generic subprogram, nongeneric package, or nongeneric subprogram declaration for P, removes subprogram P (and its subunits) from the library.

Implementation Guideline: To check that P's body has been removed from the library, attempt to compile a subunit for P. The attempted compilation should fail.

Check that if a generic package or a nongeneric package named P has been compiled, subsequent compilation of a subprogram body or declaration removes the package declaration from the library and declares a new subprogram.

Implementation Guideline: Check that the package has been removed by attempting to compile the package body.

T21. Check that any of the predefined library units can be recompiled, making the predefined units inaccessible (see IG 8.6/T1).

T22. Check that if a subprogram body is initially compiled with a context clause and a unit named in the context clause is recompiled, then an attempt to compile the body again will succeed if the context clause is present.

Check that the recompilation of the body will fail if the body refers to the recompiled unit but the unit is not named in the body's context clause.

Check that if the recompiled unit is not needed in the subprogram body, the body can be successfully recompiled without mentioning the recompiled unit.

10.1.1 With Clauses

Semantic Ramifications

S1. Since a with clause refers to a unit implicitly declared in STANDARD prior to the unit being compiled (RM 8.6/2), a with clause cannot contain the name of a library unit being compiled, e.g.,

```
with P;
package P is    -- illegal; two implicit declarations of P
```

But a secondary unit can name its library unit, since the secondary unit does not redeclare the library unit named in the with clause:

```
with P;
package body P is
```

S2. The library unit ancestor of a subunit can also be mentioned in a with clause:


```

with P;
separate (P.Q.R)
package body S is

```

S3. The names in a use clause must have been mentioned in the preceding with clauses (RM 10.1.1/3); any package name in any preceding with clauses can be mentioned. Although the with clauses given for a compilation unit apply to its subunits, a use clause cannot mention the name of a package appearing in any ancestor's with clause; the RM limits the names to those appearing in the same context clause. Finally, the rule specifying what package names can appear in a context specification's use clause excludes package names whose visibility is achieved by direct declaration instead of by the effect of the preceding with clauses. The following examples illustrate these points:

```

package B is
    type TB is ...
end B;

package A is
    package PA is
        type TA is ...
    end PA;
    package RR is ... end RR;
end A;

with A;                                -- (1)
package P is
    X : A.PA.TA;
end P;

with B; use A;                          -- (2); illegal use of A
package body P is
    use A;                              -- legal use of A
    package QQ is
        type TQ is ... ;
    end QQ;
    package RR is ... end RR;           -- hides A.RR
    use QQ, PA;
    procedure PP (X : TA; Y : B.TB; Z : TQ) is separate;
end P;

with B; use B;                          -- (3)
separate (P)
procedure PP (X : TA; Y : TB; Z : QQ.TQ) is
    use QQ;
    ZZ : TQ;

```

The with clause at (3) is required, because a use clause cannot be written for B unless a preceding with clause names B. In addition, note that you cannot write use B, QQ at (3) since the RM says that the only package names permitted in a use clause of a context specification are those made visible by previous with clauses of the same context specification. The with clause at (2) doesn't count at (3); the context clauses are independent. Note that the effect of use clauses at the stub for PP carry over to the declaration of the subunit PP.

S4. A name denoting a library unit is not allowed in a context clause if the name has been declared by a renaming declaration:

```

package P is
  package Q renames P;
end P;

with P; use P;
with Q;      -- illegal
...

```

Although Q is the visible name of a library unit, it is not allowed in a with clause or a use clause of a context clause.

S5. Although a unit named in a with clause is implicitly declared in STANDARD, an expanded name denoting the unit, e.g., STANDARD.P cannot be used within a subsequent use clause of the context clause.

S6. In effect, the only names visible in a with clause are names of library units.

```

package P is
  package R is
    X : INTEGER := 5;
  end R;
end P;

package R is
  X : INTEGER := 7;
end R;

with P; use P;      -- P.R is directly visible now
with R;             -- only library unit names visible in with clauses
use R;              -- library unit R
procedure S is
  Y : INTEGER := X;  -- STANDARD.R.X

```

Since no use clause is given for P.R, P.R.X is not directly visible.

Changes from July 1982

S7. A simple name declared by a renaming declaration is not allowed in a context clause.

Changes from July 1980

S8. A sequence of use clauses is allowed in a context clause.

S9. Names in the with clauses and use clauses of a context clause must be simple names.

S10. The with clauses and use clauses of a context clause given for a package specification, a subprogram declaration, or a generic unit also apply to the corresponding secondary unit.

Legality Rules

L1. The names appearing in with clauses can only be simple names of (previously compiled) library packages, subprograms, and generic units.

L2. The use clause of a context clause can only name library units named in the preceding with clauses of the context clause.

Test Objectives and Design Guidelines

T1. Check that

- a. the names in a with clause cannot be the names ASCII, TRUE, or FALSE (when there is no library unit named TRUE or FALSE), nor can the name in a use clause be ASCII.
- b. a subunit or a nested package cannot be named in a with clause or use clause.
- c. a library unit, P, cannot be named in a with clause or use clause as STANDARD.P, nor can such a library unit be named in a use clause if it has not appeared in a preceding with clause.
Implementation Guideline: Try one case where the name in the use clause appeared in a with clause that is applicable to the unit being compiled, e.g., give with P for a package specification, and then for the body, try writing use P. Try this for subunits as well.
- d. with STANDARD; is not permitted if there is no user-defined library unit called STANDARD.
- e. the name of a package specification, subprogram declaration (generic or nongeneric), and subprogram body (for the first compilation) cannot be the same as a name in a with clause attached to the unit.
- f. a use clause cannot name a subprogram mentioned in a preceding with clause.
- g. a use clause cannot mention a name made visible by a preceding use clause in the same context specification.

T2. Check that more than one with clause can appear in a context specification.

Check that two or more use clauses can appear in succession in a context clause.

T3. Check that a with clause provides access to previously compiled subprogram and package library units (implicitly checked by other tests).

T5. Check that a with clause can name a library unit specified in the same or a previous with clause.

Check that a with clause associated with a subunit can name its ancestor library unit as well as a library unit specified in a with clause of any ancestor unit.

T6. Check that a with clause for a package body (generic or nongeneric) or for a generic subprogram body can name the corresponding specification, and a use clause can also be given.

Check that a with clause for a subprogram body can name the subprogram if a declaration for the subprogram already exists in the library.

Implementation Guideline: Either compile the subprogram twice or first compile an explicit declaration for the subprogram.

T7. Check that if a with clause in a subunit contains a name, e.g., RR, that has been locally declared in one of the subunit's ancestors, the with clause's name is nonetheless considered to refer to a library unit, and the library unit can be accessed as STANDARD.RR.

T8. Check that a with clause and a use clause given for a package specification applies to the body and the subunits of the body.

Implementation Guideline: Include a test for subunits of subunits.

Check that if with clauses are given both for a specification and a body, and the with clauses name different library units, the library units named in all the with clauses are visible in the body and its subunits.

- T9. Check that a name declared by a renaming declaration cannot be used in a with clause or in a use clause of a context clause.

Implementation Guideline: The name should denote a library unit named in an earlier with clause.

- T10. Check that a use clause for a subunit cannot name a package that is only named in a with clause of an ancestor unit.

10.2 Subunits of Compilation Units

Semantic Ramifications

- S1. RM 10.2/3 says "A body stub is only allowed as the body of a program unit ... if the body stub occurs immediately within either the specification of a library package or ..." Since a body stub is not syntactically allowed in a package specification, the part of the rule mentioning package specifications can be ignored.

- S2. Since subunits having the same ancestor library unit must have distinct names, body stubs cannot introduce two or more overloaded body stubs within the same declarative part.

- S3. Subunits of different parents having a common ancestor cannot have identical identifiers as their names, even though the expanded name of the subunits' parents will always be distinct:

```

procedure P is
  procedure P1 is separate;
  procedure P2 is separate;
begin ... end P;

separate (P)
procedure P1 is
  procedure NOT_UNIQUE is separate;
begin ... end P1;

separate (P)
procedure P2 is
  procedure NOT_UNIQUE is separate;  -- illegal
begin ... end P2;

```

The declaration of NOT_UNIQUE is illegal since P.P1.NOT_UNIQUE and P.P2.NOT_UNIQUE have a common ancestor unit, P.

- S4. Since separately compiled subprograms cannot have operator symbol names (RM 10.1/3), no subprogram whose designator is an operator symbol can have its body specified with a body stub.

- S5. Because subunits are compiled as though they were substituted for the corresponding body stub, an implementation must be careful to ensure that entities declared after a body stub are not visible within the body for the stub:

```

package P is
  Y : INTEGER := 6;
end P;

package body P is
  X : INTEGER;
  procedure Q is separate;

```

```

package P is
  Y : INTEGER := 5;
end P;
procedure R is separate;
end P;

separate (P)
procedure Q is
  Z : INTEGER := P.Y;      -- not P.P.Y
begin ... end Q;

separate (P)
procedure R is
  Z : INTEGER := P.Y;      -- P.P.Y

```

Within Q, P.P is not visible, but P.P is visible within R, so within procedure R, P.Y refers to P.P.Y.

s6. RM 10.2/8 notes that although two subunits cannot be overloaded, renaming can be used to achieve overloading, e.g., as follows:

```

package S is
  procedure Q1 (A : INTEGER);
  procedure Q2 (B : FLOAT);
end S;

with S; use S;
package P is
  procedure Q (A : INTEGER) renames Q1;
  procedure Q (B : FLOAT) renames Q2; -- overloading introduced
end P;

package body S is
  procedure Q1 (A : INTEGER) is separate;
  procedure Q2 (B : FLOAT) is separate;
end P;

```

s7. The order of subunit elaboration is determined by the order in which body stubs appear in a parent. Therefore, any initializations caused by the subunit elaborations take place in the order specified by the body stubs. For example:

```

package body P is
  X : INTEGER;
  package body Q is separate;
  package body R is separate;
end P;

separate (P)
package body R is
begin
  X := 5; -- P.X
end R;

separate (P)
package body Q is

```

```

begin
    X := 6;    -- P.X
end Q;

```

Even if these three units are submitted together in a single compilation, after P is elaborated, P.X has the value 5, since package body R must be elaborated after package body Q. This example also shows that subunits do not have to be compiled in the order of their body stub declarations.

s8. RM 10.2/4 requires that the subprogram specifications for a body stub and the corresponding proper body conform as specified by RM 6.3.1. RM 6.3/3 requires that the specification given in a subprogram declaration conform with that given in a corresponding body stub. Because conformance is defined between pairs of subprogram specifications, it is possible for the declaration and the proper body of a subprogram to not satisfy the conformance rules even though the rules are satisfied for the declaration/stub and stub/proper-body pairs:

```

package A is
    type T is (ONE, TWO);
end A;

with A; use A;
package P is
    procedure Q (X : A.T);           -- 1
end P;

package body P is
    package B renames A;
    procedure Q (X : T) is separate; -- 2
begin ... end P;

separate (P)
procedure Q (X : B.T) is           -- 3
begin ... end;

```

The specifications at 1 and 2 conform, and so do those at 2 and 3, but 1 and 3 do not conform, since A and B are not declared by the same declaration (see IG 6.3.1/S).

s9. A generic unit can be instantiated before its body can be compiled since its body can be given as a stub:

```

procedure P is
    INT : INTEGER := 1;
    FLT : FLOAT   := 1.0;

    generic
        type T is private;
        C : in T
        V : in out T;
    package GP is
    end GP;

    package body GP is separate;

```

```

package GP1 is new GP (INTEGER, 2, INT);
package GP2 is new GP (FLOAT, 3.0, FLT);

begin
    null;
end P;

separate (P)
package body GP is
begin
    V := C;
end GP;

```

The effect of the GP1 instantiation is to change the value of INT to 2; the GP2 instantiation changes FLT to 3.0.

S10. The library unit named in a subunit's context clause need not be directly visible within the subunit because it can be hidden by a declaration in some parent unit:

```

procedure P is                -- library unit
    LIB : INTEGER;            -- will be name of library unit
    procedure PS is separate;
begin ... end P;

procedure LIB is              -- library unit
begin ... end LIB;

with LIB;
separate (P)
procedure PS is
begin
    LIB;                      -- illegal; only P.LIB is visible
    STANDARD.LIB;             -- ok
    LIB := 4;                 -- ok
end PS;

```

S11. Similarly, the use clauses for a subunit are combined with the use clauses of parent units to determine which identifiers are visible within a subunit:

```

package P1 is
    INT : INTEGER;
end P1;

package P2 is
    INT : INTEGER;
end P2;

with P1; use P1;
procedure PR is
    procedure PS is separate;
begin
    ...
end PR;

```

```

with P2; use P2;
separate (PR)
procedure PS is

```

```

    X : INTEGER := INT; -- illegal
begin
    ...
end PS;

```

The use of INT is illegal because INT is not directly visible. The use clause for P1 and the use clause for P2 are both in effect throughout PS's body, and since INT is declared in both P1 and P2 (and overloading is not allowed for both declarations), neither INT declaration is made directly visible (RM 8.4/6).

Changes from July 1982

S12. The rule stating where body stubs are allowed now refers to where the body stub appears rather than to where the declaration of the program unit being stubbed occurs.

Changes from July 1980

S13. The effect of context clauses on visibility within proper bodies has been clarified.

Legality Rules

- L1. A body stub is permitted only in the outermost declarative part of a package body, subprogram body, or task body serving as a compilation unit (RM 10.2/3). (Note: a task body can be a compilation unit only if it is a subunit.)
- L2. The subprogram specification given in a body stub must conform with that given for the proper body (RM 10.2/4).
- L3. If a body stub is given as the body corresponding to a subprogram declaration, the specification in the declaration and stub must conform (RM 6.3/3).
- L4. A subunit must give the expanded name of the parent unit, i.e., the first identifier in the unit name must be the simple name of a library unit and the last identifier must be the name of the unit containing the body stub for the unit being compiled (RM 10.2/5).
- L5. Two body stubs in the same declarative part cannot declare the same identifier (RM 10.2/5).
- L6. The designator in a body stub must not be an operator symbol (RM 10.2/5, RM 10.1/3).
- L7. Subunits having a common ancestor unit must be named with distinct identifiers (RM 10.2/5).
- L8. A body stub for a package or task body is forbidden if a corresponding package or task declaration has not appeared either earlier in the same declarative part (RM 3.9/9) or in the corresponding package specification (RM 7.1/4) (when the declarative part is for a package body).
- L9. Declarations appearing after a body stub are not visible within the proper body for the stub (RM 10.2/6).

Test Objectives and Design Guidelines

- T1. Check that
 - a. a body stub cannot be given in a declarative part enclosed by another declarative part, e.g., in a procedure nested in a package body.

- b. a task or package body stub cannot be given if there has been no preceding specification for the stub.
- c. two body stubs in the same declarative part cannot declare overloaded subprograms.
- d. a body stub cannot be given for a subprogram whose designator is an operator symbol.
- e. two body stubs cannot be declared with the same identifier if the stubs have a common ancestor,
Implementation Guideline: Check when the stubs are in different declarative parts but have the same parent unit (which need not be a library unit).
- f. if the identifier of a subunit is unique in a program library and it is the parent of another subunit, the unit name following `separate` cannot be just the name of the parent.
- g. when compiling a subunit of library unit P, check that `STANDARD.P` cannot be used as the name following `separate`.
- h. if a body stub is deleted from a compilation unit, the previously existing subunit can no longer be accessed (see also IG 10.1/T20).

- T2. Check that subunits having different ancestor library units can have the same name.
- T3. Check that identifiers declared prior to a subunit's body stub are visible in the subunit, but those declared after the body stub are not.
- T4. Check that identifiers declared in ancestors of a subunit are visible within the subunit.
Implementation Guideline: Try at least grandparents of the subunit.
Implementation Guideline: Check the effect of use clauses appearing in the grandparent and parent.
- T5. Check that a `with` clause associated with a subunit can name a library unit specified in a `with` clause of any ancestor unit (see IG 10.1.1/T5).
- T6. Check that if a `with` clause for a subunit contains a name that is directly visible in one of the subunit's ancestors, the `with` clause name is nonetheless considered as referring to a library unit, not the locally declared entity (see IG 10.1.1/T7).
- T7. Check that subunits are elaborated in the order in which their body stubs appear, not (necessarily) in the order in which they are compiled.
- T8. Check that if an overloaded subprogram is declared, one of the subprogram bodies can be specified with a body stub and compiled separately.
- T9. Check that a generic subunit can be specified and instantiated.
Implementation Guideline: Include a check that the unit can be instantiated before the body stub occurs.
Implementation Guideline: There are two cases: 1) when the parent and the subunit are in the same compilation file, and 2) when they are not.
- T10. Check that subunit names can be identical to identifiers declared in `STANDARD`, namely, `BOOLEAN`, `INTEGER`, `FLOAT`, `CHARACTER`, `ASCII`, `NATURAL`, `POSITIVE`, `STRING`, `DURATION`, `CONSTRAINT_ERROR`, `NUMERIC_ERROR`, `PROGRAM_ERROR`, `STORAGE_ERROR`, and `TASKING_ERROR`.
- T11. Check that a body stub must conform to a preceding declaration, and a proper body must conform with its stub.

Implementation Guideline: Check for a mixture of generic and nongeneric units.

Check that for a subprogram declaration-stub-body triple, the declaration-stub and stub-body specifications can conform but the declaration-body specifications need not.

T12. Check that a body stub can serve as an implicit declaration of a subprogram (RM 6.3/3), i.e., a preceding subprogram declaration is not required.

T13. Check that the use clauses given for ancestors of a subunit are combined with any use clause given for a subunit itself.

Check that an identifier declared in a library package named in a use clause is not visible in a subunit if the identifier is declared in a parent unit and is directly visible at the stub.

10.3 Order of Compilation

Semantic Ramifications

S1. If a file being compiled does not satisfy the syntax for a "compilation" (see RM 10.1/2), the whole file can be rejected:

```

procedure P;

procedure Q;          -- programmer meant to write is instead of ;
begin
    ...
end Q;

procedure R;

```

After processing this file, the library need not contain P, Q, or R because the sequence of lexical elements does not satisfy the syntax for a compilation. But RM 10.4/1 says language rules are enforced in the same manner whether compilation units are submitted as independent compilations or together in a single compilation. And RM 10.4/2 says the library file "is updated for each compilation unit successfully compiled." Finally, RM 10.1/7 says "the compilation units of a compilation are compiled in the given order." These statements together imply that the intent of the RM is to at least allow an implementation to process a compilation as a sequence of compilation units, optionally followed by lexical elements that do not satisfy the syntax for a compilation unit. Hence, if a syntax error is discovered after processing an initial sequence of (well-formed) compilation units, the successfully processed units can be added to the library. In the above example, this reasoning implies that it is permitted to add procedure P to the library despite the existence of a syntax error. It is even permitted to add procedure Q to the library since "procedure Q;" is a legal declaration of Q. Finally, if an implementation's syntax error recovery mechanism allows procedure R's declaration to be identified as a compilation unit, this subprogram can also be added to the library.

S2. If an error is internal to a subprogram compilation unit, then the unit is not added to the library. For example,

```

procedure PP;

procedure QQ is
begin
    if TRUE() then      -- syntax error
        null;
    end if;
end QQ;

procedure RR;

```

If compilation units are processed one at a time, it is likely that despite this syntax error, most implementations will correctly detect the end of procedure QQ, and will continue to successfully process procedure RR. Such an effect is allowed and desirable so the maximum number of errors can be detected in a single compilation submission. To detect whether procedure RR has been added to the library, a later compilation unit or a compilation can contain:

```
with RR;
procedure SS is
begin
    RR;
end SS;
```

Such a compilation cannot be compiled successfully unless RR has been added to the library. Note that no unit dependent on QQ can be added to the library, since QQ contains an error (RM 10.3/3).

S3. RM 10.3/3 says that if "any error" is detected while attempting to compile a compilation unit, at least that unit must be rejected. The phrase "any error" can encompass errors not necessarily in the unit being compiled. In particular, compiling or recompiling a generic unit body may imply that an illegal instantiation exists in a previously compiled unit:

```
generic
    type T is private;
package GP is
    procedure PR (X : T);
end GP;

with GP;
procedure M is
    package NGP is new GP (STRING);
begin
    ...
end M;

package body GP is
    X : T;           -- potentially illegal
    procedure PR (X : T) is
    begin ... end PR;
end GP;
```

The instantiation of GP is illegal given that GP's body declares a variable of type T (RM 12.3.2/4). However, at the time M is compiled, the instantiation is legal since the package specification contains no uses of the formal type T that are illegal when the actual parameter is STRING. The package body, GP, is also, when considered only in conjunction with its specification, perfectly legal — the use of type T within the body is legal, and the declaration of GP.PR satisfies all the rules.

S4. An implementation has two choices in this situation:

- it can reject GP's body;
- it can accept GP's body, but refuse to execute M as a main program.

It can reject GP's body on the basis that if GP's body were to be accepted, the instantiation in M would be illegal, i.e., when compiling GP, an "error is detected" in M.

S5. On the other hand, the compiler can accept GP's body because the body, when considered

independently of instantiations, is quite legal. Of course, if GP's body is accepted, the instantiation in M is still illegal and M's execution as a main program cannot be allowed. If an attempt is made to execute M, a link-time error message should be given indicating the presence of an illegal instantiation in M. Such a message must be issued before any library unit is elaborated. (Detection of errors at link time satisfies the RM's requirement to detect the illegal instantiation at "compile time" (RM 1.6/2-3), since compile time only ends when execution of the main program begins; execution includes elaboration of library units used by the main program.) In no case can the previously compiled unit, M, be removed from the library.

S6. If GP's body is accepted for compilation and is added to the library, then it must be possible to compile a replacement for this body that does not contain any object declaration for type T (or for any other uses of T that would make M's instantiation illegal). It must then be possible to execute M as a main program without recompiling M.

S7. The recompilation of a generic unit body is not allowed to make obsolete any units that have instantiated the generic unit. RM 10.3/6 says that a change to a subprogram or a package body does not affect other compilation units (apart from subunits of the body). This rule applies to generic as well as nongeneric units, since the syntactic terms "package body" and "subprogram body" are used in the rule (see RM 1.5/6), and a package_body or subprogram_body can be used for either a generic or a nongeneric unit. The rule has important implementation consequences. Consider the following units:

```
generic
  type T is range <>;
  procedure G (X : T);

with TEXT_IO; use TEXT_IO;
procedure G (X : T) is
  package TIO is new INTEGER_IO(T);
  use TIO;
begin
  PUT (X + X*5);
  NEW_LINE;
end G;

with G;
procedure M is
  procedure INT is new G(INTEGER);
  procedure LNG is new G(LONG_INTEGER);
begin
  INT (5000);
  LNG (5000);
end M;
```

Now suppose we replace G's body with:

```
with TEXT_IO; use TEXT_IO;
procedure G (X : T) is
  package TIO is new FLOAT_IO(FLOAT);
begin
  PUT(FLOAT(X) + FLOAT(X)*5.0);
  NEW_LINE;
end G;
```

After compiling the new body for G, it must be possible to execute M as a main procedure

without recompiling M. Note that for most implementations, the two instantiations of G in M will require different code bodies since different machine instructions will be used for the arithmetic operations, and the instantiations of INTEGER_IO will require different bodies as well. When the new body for G is compiled, the code bodies for the old G must be discarded, and the calls to INT and LNG must be linked to two new code bodies generated for the new version of G. Note also that it is immaterial whether G's original body is compiled before or after M — both sequences of compilations are legal and have the same semantic effect.

S8. Of course, G's new body can occur in the same compilation file as the original compilation, so an implementation must be prepared to cope with the recompilation of generic unit bodies even if generic bodies and specifications are required to be in the same compilation file (RM 10.3/9).

S9. The situation is no different for generic packages:

```
generic
  type T is range <>;
  VAL : T;
package GG is
  X : T := VAL;
end GG;

with GG;
procedure MM is
  package INT is new GG(INTEGER, 5000);
  package LNG is new GG(LONG_INTEGER, 5000);
begin
  if LONG_INTEGER(INT.X) /= LNG.X then
    ... -- print error message
  end if;
end MM;
```

As written, MM can be executed immediately as a main program, but it would also be possible to compile a package body for GG after compiling MM. Such a body could perform operations on X. For example, a procedure body for G, ... could print out a function of X. If a body for GG is provided after compiling MM, it must then be possible to execute MM without recompiling MM. Finally, the package body for GG could be recompiled, and MM again executed without first having to be recompiled.

S10. If the INLINE pragma is not supported by an implementation (see RM 6.3.2/4), then no dependence is created on a body.

S11. Interprocedural optimizations (i.e., optimizations whose validity depends on knowledge of the body of a called subprogram) are only allowed when the subprogram body and units containing calls to the subprogram are processed together in the same compilation file (RM 10.3/8). Optimizations in general are not allowed to create dependences on bodies such that if a body is changed, code in a calling program will be incorrect.

S12. If an implementation requires that generic declarations and bodies be compiled together in the same compilation, then this restriction must be enforced uniformly. It cannot be limited to just those generic units having, say, formal private type parameters. An implementation must similarly be consistent in its requirement for compiling generic subunits together with their parent units.

S13. The set of units potentially affected by the recompilation of a unit is determined in a transitive manner, since "potentially affected" is a transitive relation. It is transitive because

marking a unit as "potentially affected" constitutes a "change" in that unit. For example, if a secondary unit is marked as "potentially affected," this is a change in the unit and so its subunits, if any, must also be marked as "potentially affected." All potentially affected units are then marked as being obsolete. An implementation can then reduce recompilation costs if it can determine that the "effect" of recompiling an obsolete unit will be the same as the effect of simply deciding that the previously compiled unit is actually not obsolete. In particular, reusing a potentially affected unit (i.e., not marking it as obsolete) is forbidden if the recompilation of the unit would be unsuccessful (e.g., because the unit is no longer legal).

S14. Suppose that a package specification requires a body and that after compiling the body, the specification is recompiled. If no new body is compiled, the old body will not be used because it is obsolete and not needed by the new package specification (RM 10.3/5):

```
package P is
  X : INTEGER := 7;
  function F return INTEGER;
end P;

package body P is
  function F return INTEGER is
  begin
    return 6;
  end F;
begin
  X := 5;
end P;

package P is          -- recompilation of P's specification
  X : INTEGER := 7;
end P;
```

The fact that a body existed for the previous version of P is not relevant. Consequently, P.X must have the value 7 when execution of the main program begins.

S15. Suppose now that we do compile a new body for P and then a new specification:

```
package body P is
begin
  X := 5;
end P;

package P is
  X : INTEGER := 6;
end P;
```

The recompilation of P's specification makes P's body obsolete. Since a body is not required for the new specification, it is not necessary to recompile the body — the main program can begin execution even if P does not replace the old body. Note that the last sentence of RM 10.3/5 (which allows an implementation to sometimes reuse a potentially affected unit without recompiling it) only applies to units that must be recompiled when they are made obsolete. Since recompilation of P's body is not required, the old body cannot be reused. In essence, the recompilation of a package specification always makes an optional package body obsolete, even if the old and new package specifications are identical.

S16. In general, adding declarations to a package specification, even at the end, can affect the legality of other units that name the package specification in a use clause. The new declarations can cause previously visible declarations to be hidden:

```

package P is
  X : INTEGER;
end P;

package Q is
  Y : FLOAT;
end Q;

with P, Q; use P, Q;
procedure M is
  F : FLOAT := Y;      -- Q.Y
begin ... end M;

```

Suppose P is now modified by adding a declaration of Y:

```

package P is
  X : INTEGER;
  Y : INTEGER;
end P;

```

After compiling the revised, P two Y's are potentially visible in M because of the use clause (RM 8.4/4). Since neither potentially visible Y is overloadable, neither is actually made visible (RM 8.4/6). So deciding whether a potentially affected unit is actually affected by a recompilation requires more than determining whether generated code will be affected — the modification may make a previously legal unit illegal as well.

S17. A package body can be made obsolete by recompiling its package specification. It can also be made obsolete if a unit mentioned in its with clause is recompiled:

```

package P is
  X : INTEGER := 5;
end P;

package Q is
  Y : INTEGER := 10;
end Q;

with P;
package body Q is
begin
  Q.Y := P.X;
end Q;

-- compile a new version of P
package P is
  X : INTEGER := 6;
end P;

```

After the compilation of the second version of P, Q's body is obsolete. Since Q's specification does not require a body, no body for Q need be recompiled before executing a main program referring to packages P and Q. In fact, it would be incorrect for an implementation to require compilation of a body for Q since Q does not require a body. (If a programmer wishes to ensure that a body is always compiled for Q, a subprogram specification or incomplete type declaration should be given in Q's private part. The presence of such a specification or type declaration means that a body is required.)

S18. If a package specification declares a task object, no package body is required, although a body is implicitly provided if the programmer fails to provide one (RM 9.3/5). Consequently, if a programmer-provided package body is subsequently made obsolete by recompiling a library package mentioned in its context clause, no recompilation of the package body is required:

```

package P is
    task type T;
end P;

package body P is
    task body T is
        ...
    end T;
end P;

with P;
pragma ELABORATE(P);
package Q is
    OBJ : P.T;           -- no body required for Q
    X   : INTEGER := 10;
end Q;

package R is
    Y : INTEGER := 5;
end R;

with R;
package body Q is        -- a body is provided for Q
begin
    Q.X := R.Y;
end Q;

-- recompile R
package R is
    Y : INTEGER := 9;
end R;

```

Since no body is required for Q, the main program can be executed without providing an explicit body for Q. An implicit null body is used instead of the old programmer-provided body.

Changes from July 1982

S19. Subunits of a generic unit can be required to be part of the same compilation as the parent unit.

Changes from July 1980

S20. The RM states more explicitly that if an error is detected while attempting to compile a unit, then the unit is rejected and not added to the library.

Legality Rules

- L1. A package specification, a subprogram declaration, or a generic unit declaration cannot be successfully compiled if any units named in its context clause are obsolete or have not been successfully compiled at least once (RM 10.3/2, /5).

- L2. A package body cannot be successfully compiled if its specification is obsolete or has not been successfully compiled at least once (RM 10.3/2, /5), or if any units named in its context clause are obsolete or have not been successfully compiled at least once (RM 10.3/2, /5).
- L3. A subprogram body cannot be successfully compiled if its declaration is obsolete, or if any units named in the body's context clause are obsolete or have not been successfully compiled at least once (RM 10.3/2, /5).
- L4. A subunit cannot be successfully compiled if its parent unit is obsolete or has not been successfully compiled at least once (RM 10.3/2, /5), or if any units named in the subunit's context clause are obsolete or have not been successfully compiled at least once (RM 10.3/2, /5).
- L5. If a library package specification, a subprogram specification, or a generic unit declaration is modified in a way that invalidates the legality or the correctness of code generated for units using the modified declaration, then all these units must be recompiled before execution of the main program is permitted (except for units that are no longer needed, i.e., package bodies for specifications that do not require a body) (RM 10.3/5).
- L6. If the parent of a subunit is modified in a way that affects the correctness of code generated for the subunit, then the subunit must be recompiled before execution of the main program is permitted (RM 10.3/5) unless the subunit is no longer needed (i.e., unless its body stub has been deleted from its parent).
- L7. If a pragma `INLINE` is obeyed and an inline subprogram body is substantively modified, all units calling the subprogram must be recompiled (RM 10.3/7).
- L8. An implementation may require that generic unit bodies (together with their subunits) be compiled together with their declarations in the same compilation file (RM 10.3/9).

Test Objectives and Design Guidelines

T1. Check that

- a. a package or subprogram declaration (generic or nongeneric) cannot be compiled if the units mentioned in its with clauses have not been compiled.
- b. a package body cannot be compiled if its specification has not been compiled.
- c. a package body cannot be compiled if any units mentioned in its with clauses have not been compiled.
- d. a subprogram body cannot be compiled if any units mentioned in its with clauses have not been compiled.
- e. a subunit cannot be compiled if its parent has not been compiled.
Implementation Guideline: Use a subunit having more than one ancestor as well as a subunit with a single ancestor.
- f. a subunit cannot be compiled if any units mentioned in its with clauses have not been compiled.

T2. Check that a nongeneric package body can be compiled after compiling a unit that refers to the package specification. (Generic units are checked in T5).

Check that the recompilation of a subprogram or package body causes the new body to be used in place of the old body.

T3. If a package specification (or a subunit parent) is changed only with respect to its

comments or formatting and is then recompiled, check whether the implementation nonetheless requires recompilation of all subordinate units.

Check whether recompiling a package specification to which declarations have been added at the end causes the recompilation of subordinate units to be required.

Implementation Guideline: Include one case where the new declarations cause another unit to become illegal because of the effect of use clauses.

- T4. Check that if the body of an `INLINE` subprogram is modified and the pragma is obeyed, all units calling that subprogram as an inline subprogram must be recompiled.

Implementation Guideline: If the `INLINE` pragma is obeyed, recompiling a body should require the recompilation of units calling the body. If it is not obeyed, no recompilation will be required and the new body will be used. (Note: failure to obey the pragma does not make a program illegal.)

Implementation Guideline: Compile the first body before any calls are compiled (since the `INLINE` pragma need not be obeyed until a body is compiled).

- T5. Check that if a separately compiled generic unit body is substantively modified, no unit instantiating it need be recompiled.

Implementation Guideline: Use a body with at least two instantiations so the instantiations cannot conveniently use the same generated code. Use instantiations of integer or real I/O as shown in IG 10.3/S in one case, but also create a case that will be effective even when an implementation does not fully support `TEXT_IO`.

Implementation Guideline: Use one case with generic subunits (see IG 10.2/T9 for a similar test).

- T6. Check that if a package specification is substantively modified (e.g., by changing declarations in a way that changes their type or size), previously compiled units using the modified declarations must be recompiled.

If a package body was provided for a package that does not require a body, and if the package specification is modified but still does not require a body, check that the original package body is no longer used.

Similarly, check that an optional package body is no longer used after recompiling a unit named in the body's context clause.

Implementation Guideline: Include a case where the package specification declares an object of a task type, but the specification does not require a body.

Implementation Guideline: Write these tests in such a way that the pragma `ELABORATE` is not required.

- T7. Check that if a subprogram declaration is substantively modified, all units invoking that subprogram must be recompiled, and so must the subprogram body.
- T8. Check that if the parent of a subunit is substantively modified, all its subunits must be recompiled.
- T9. Check that when compiling a subprogram library unit body or declaration, no declaration for the subprogram is added to the library if an error is found.
- T10. If a generic unit can be separately compiled for a nonprivate formal type, check that it can also be separately compiled when the formal type is private.
- T11. Check whether an implementation requires generic unit bodies and subunits to be compiled together in the same file (see IG 10.1/T12).
- T12. Check that if a subprogram body is compiled with a context clause and some unit mentioned in the clause is recompiled, an explicit subprogram declaration need not be recompiled before the subprogram body can be recompiled (see IG 10.1/T22).
- T13. Check that units indirectly affected by a recompilation are considered obsolete.

10.4 Program Library

Semantic Ramifications

S1. Commands for reusing units of other program libraries could permit an implementation to provide a way of adding units to a program library without submitting them first for compilation. Such a facility might be used to add application-oriented packages to a particular library of Ada programs. Or, if the program library interfaces with a version control system, these commands might be used to indicate which versions of program units are to be used when compiling some unit or when initiating the execution of a main program.

S2. The commands for interrogating the status of program library units could be used to indicate which units require recompilation, which units are missing, which units are unreferenced, etc.

Changes from July 1982

S3. The program library is updated only for the successful compilation of a compilation unit.

Changes from July 1980

S4. The possible existence of commands for dealing with program libraries has become a note.

Test Objectives and Design Guidelines

T1. Check that the library file keeps appropriate information for checking the consistency of separately compiled units (implicitly checked by all tests for separate compilation).

10.5 Elaboration of Library Units

Semantic Ramifications

S1. Only legal programs can be executed. Consequently, before attempting to elaborate library units required by a main program, an implementation must make sure that all required units are present, e.g., that all packages for which bodies are required indeed have bodies, that all subprogram declarations have bodies, that subunits are present for all parent units, and that all units mentioned in context clauses are present. A required unit can fail to be present either because it was never compiled (only for bodies) or because it is obsolete as a result of some other unit's compilation (for bodies and declarations). In addition, the library units referenced directly or indirectly as a result of the main program's context clause must be present whether or not the units are used by the main program. For example:

```

procedure P is
begin
    ...
end P;

with P;
procedure MAIN is
begin
    if FALSE then
        P;
    end if;
end MAIN;

procedure P (X : INTEGER := 1);

```

After compiling the new declaration for P, the main program cannot be executed since no body is present for the new version of P and a body is required. Even though a compiler can determine that P is never called in the main program, a body must be provided since P is mentioned in MAIN's with clause.

S2. Except for the main program, if a library unit Q is named in the context clause for unit R, Q's body need not be elaborated before R is elaborated. Pragma ELABORATE is the only way to ensure that Q's body is elaborated first.

S3. It is not possible for an implementation to always elaborate a library unit's body immediately after elaborating the unit's specification since the body might mention new library units in its context clause:

```
package P is
...
end P;

with Q, R;
package body P is
...
end P;
```

P's body cannot be elaborated until the specifications of Q and R have been elaborated.

S4. Elaboration order circularities can arise by using pragma ELABORATE:

```
package A is
...
end A;

package B is
...
end B;

with A;
pragma ELABORATE (A);
package body B is
...
end B;

with B;
pragma ELABORATE (B);
package body A is      -- circularity
...
end A;
```

The circularity can only be detected after seeing the name of the unit being compiled. Some implementations may defer detection of such circularities until an attempt is made to execute the main program (when an implementation is determining the required elaboration order for all units). Since only legal programs can be executed, the circularity must be detected and the program rejected before any library units are elaborated.

S5. In addition, circularities can arise in other ways:

```
package G is
end G;
```

```

package body G is
end G;

with G;
pragma ELABORATE (G);
procedure USE_G;

with USE_G;          -- circularity introduced here
package body G is
end G;

```

The pragma requires that G's body be elaborated before USE_G's declaration is elaborated. The with clause for USE_G requires that USE_G's declaration be elaborated before G's body (since the with clause is associated with G's body). Hence, an illegal circularity has been introduced.

S6. The RM says the library unit named in the pragma ELABORATE "must have a body" (RM 10.5/4). Although "must have a body" seems to impose a legality condition on the use of the pragma, RM 2.8/9 says a pragma "has no effect if ... its arguments do not correspond to what is allowed for the pragma." Hence, if no body is actually provided, the pragma must have no effect (i.e., a warning message can be printed, but execution of the main program cannot be prevented just because an optional package body is not present).

S7. If an implementation chooses to warn users when a unit named in a pragma ELABORATE does not have a body, the warning must be deferred until an attempt is made to execute the main program. For example, even if a body is present when a unit is compiled, the body can later be made obsolete, so a check for the presence of a body cannot in general be made until link time:

```

package A is
...
end A;

with P;
package body A is
...
end A;

with A;
pragma ELABORATE (A);
procedure M is
...
end M;

-- recompile P, making A's body obsolete

```

When P's specification is recompiled, A's body becomes obsolete (see IG 10.3/S and RM 10.3/5). When procedure M was compiled, A had a body. However, if P is recompiled, A's body is made obsolete, and the recompilation rules do not require that A's body be recompiled if A's specification does not require a body, so the lack of a body can only be determined at link time.

S8. There is one case in which a package body is implicitly provided for a package, namely, when a package specification declares an object of a task type but no explicit body is provided. In this case, an implicit body is provided (RM 9.3/5). When a pragma ELABORATE names such a package, the implicit body must be elaborated, causing the tasks to be activated.

S9. The fact that a package specification or body calls a subprogram in another library unit does not imply that the unit's body must be elaborated first:

```

package A is
  function F return INTEGER;
end A;

package B is
  function G return INTEGER;
end B;

with A;
package body B is
  X : INTEGER := A.F;
  function G return INTEGER is ... end G;
end B;

with B;
package body A is
  Y : INTEGER := B.G;
  function F return INTEGER is ... end F;
end A;

```

In this case, if package body B is elaborated before package body A, then PROGRAM_ERROR will be raised because F's body will not yet have been elaborated. Similarly, if package A is elaborated before package B, PROGRAM_ERROR will be raised because function G's body has not yet been elaborated. The above sequence of compilation units is legal -- no circularity exists in the required elaboration order; either A or B's body can be elaborated first. An implementation must raise PROGRAM_ERROR for either elaboration order. If pragma ELABORATE is specified for both bodies, however, a circularity will be created and the program must be rejected.

Changes from July 1982

S10. The pragma ELABORATE can now accept a list of simple names.

S11. It is stated explicitly that the elaboration order for library units is not fully defined by the language.

Changes from July 1980

S12. A library unit named in the context clause of a subunit must be elaborated before the subunit's ancestor library unit.

S13. The elaboration order of library units is now determined only by the partial ordering defined by with clauses instead of by all the dependence relations resulting from the elaboration of the bodies.

S14. The pragma ELABORATE has been introduced.

S15. A program relying on the order of elaboration of library units is no longer erroneous.

Legality Rules

L1. A main program is illegal if any unit it needs is obsolete or absent. (A compilation unit *needs* all the library units named in its context clause. A package specification that requires a body (see RM 7.1/4) *needs* a package body. A subprogram declaration or a generic subprogram specification *needs* a body. A secondary unit *needs* any subunits it

declares. Finally, if a compilation unit needs unit X, and unit X needs unit Y, then the compilation unit also needs unit Y.)

- L2. A main program is illegal if the effect of a pragma ELABORATE is to require a circular order of elaboration.

Test Objectives and Design Guidelines

- T1. Check that a main program cannot be invoked if the use of pragma ELABORATE requires a circular order of elaboration.

Implementation Guideline: The illegality can be detected either at compile time or at link time.

- T2. Check that when several compilation units are submitted to a compiler in a single compilation, the order of compilation need not be the same as the required order of elaboration.

- T3. Check that the elaboration of library units required by a main program is performed consistently with the partial ordering imposed by the compilation order rules.

Implementation Guideline: In particular, check that a library unit mentioned in the with clause of a subunit is elaborated prior to the body of the ancestor library unit (and hence, prior to the body of the subunit's parent).

- T4. Check that if pragma ELABORATE is applied to a package that declares a task object, the implicit package body is elaborated (and hence, the tasks are activated).

Check that pragma ELABORATE is accepted and obeyed even if the unit named in the pragma does not yet have a body in the library, or if its body is obsolete.

Check that more than one name is allowed in a pragma ELABORATE.

- T5. Check whether an implementation elaborates library unit bodies as soon as possible after the elaboration of their specification, or whether it defers elaboration until as late as possible.

Implementation Guideline: Try a case where if the body is elaborated as soon as possible, no PROGRAM_ERROR will be raised. Include library packages, subprograms, and generic units in separate tests.

- T6. Check that a program is not rejected just because there is no way to elaborate secondary units so PROGRAM_ERROR will be avoided.

Implementation Guideline: Do not use pragma ELABORATE in this test.

- T7. Check that the execution of a main program is not allowed if a needed unit is obsolete or not yet compiled. In particular, check for obsolete library unit declarations and missing or obsolete unit bodies (including subunits). Check for generic and nongeneric units.

Implementation Guideline: Include a case where optimization might indicate that a particular library unit is actually not needed.

Check that if a package body is made obsolete but is not required by its package specification, execution of the main program is allowed (even if pragma ELABORATE is specified for the package).

Implementation Guideline: See IG 10.3/T6 for the test without a pragma ELABORATE. (A version of this test can be used here with a specification of pragma ELABORATE.)

Check that a unit named in a context clause must be present in the library even if the unit is not otherwise referenced.

- T8. Check that pragma ELABORATE is ignored if any of the names given in the pragma do not appear in a preceding context clause.

Implementation Guideline: Note: it is only possible to check that the test program is not considered illegal; an implementation is allowed to obey the pragma for those names given in the pragma that have appeared in a preceding context clause.

10.6 Program Optimization

Semantic Ramifications

S1. There is nothing in this section not covered more precisely by RM 11.6.

Changes from July 1982

S2. There are no significant changes.

Changes from July 1980

S3. There are no significant changes.

Chapter 11

Exceptions

11.1 Exception Declarations

Semantic Ramifications

S1. Exception names in Ada follow Ada's normal scope and visibility rules (see RM 8.2 and RM 8.3). Hence, a local exception can be declared using the same identifier as a predefined exception, and the local exception is distinct from the predefined exception.

S2. The RM says (RM 11.1/3) that exceptions associated with different generic instantiations are different:

```
generic
package GEN_EXC is
    EXC : exception;
end GEN_EXC;

package GP1 is new GEN_EXC;
package GP2 is new GEN_EXC;
```

GP1.EXC is distinct from GP2.EXC, which implies that

```
when GP1.EXC | GP2.EXC => ...
```

is legal, since the denoted exceptions are different (RM 11.2/5).

S3. It is necessary to distinguish the *instantiation* of a generic unit, which is a textual occurrence of a syntactic form, from the *elaboration* of an instantiation. The difference is illustrated by the following example (adapted from a comment made by G. Dismukes, February, 1981):

```
with TEXT_IO; use TEXT_IO;
procedure RAISE_EXC (CALL_AGAIN : BOOLEAN) is
    package GP is new GEN_EXC;
begin
    if CALL_AGAIN then
        begin
            RAISE_EXC (CALL_AGAIN => FALSE);
        exception
            when GP.EXC =>
                PUT_LINE ("SAME");
        end;
    else
        raise GP.EXC;
    end if;
end RAISE_EXC;
```

-- (1)

-- (2)

-- (3)

-- (4)

This procedure is called as follows:

```
begin
    RAISE_EXC (CALL_AGAIN => TRUE);
exception
```

-- (5)

```

    when others =>
        PUT_LINE ("DIFFERENT");
    end;
-- (6)

```

The instantiation at (1) will be elaborated twice, once when RAISE_EXC is called from (5) and once when RAISE_EXC is called recursively at (2). After the second call, GP.EXC is raised at (4); in the execution of (4), GP denotes the package resulting from the second elaboration of GP's declaration. This exception is passed to the handler at (3), which names the exception declared within the package resulting from the first elaboration of GP's declaration. Since there is but one instantiation, the exception name given in the handler at (3) denotes the same exception as that given in the raise statement at (4), even though the name GP denotes different packages in both these occurrences. Therefore, this program prints "SAME". The effect is the same if an explicit package declaration is written in place of the generic instantiation.

S4. If an exception is declared in the visible part of a library package, the exception can be named by separately compiled units that name the package in a context clause. An implementation must ensure that different units referring to the library package treat the package's exception as the same exception (see RM 7.2 and RM 8.3).

S5. It is possible for an exception to be propagated out of the scope of its declaration (e.g., into a subprogram that is unable to name the exception explicitly) and then have the exception propagated back into its scope (see IG 11.4/S4). This means that in general, each declared exception (even those declared in separately compiled units) must have a unique run-time identification.

Changes from July 1982

S6. PROGRAM_ERROR is raised if execution reaches the end of a function.

S7. NUMERIC_ERROR can be raised when an implementation uses a predefined numeric operation to implement some construct.

Changes from July 1980

S8. Exceptions declared in generic units are distinct for each instantiation.

S9. SELECT_ERROR is no longer a predefined exception.

S10. PROGRAM_ERROR now covers the situation SELECT_ERROR used to cover as well as other exception situations.

S11. The elaboration of a declarative item and the execution of a subprogram call now raise STORAGE_ERROR when storage is insufficient for those actions.

S12. The cases when an implementation is not required to support NUMERIC_ERROR are listed in RM 4.5.7/7 and RM 3.5.4/10.

Test Objectives and Design Guidelines

T1. Check that the predefined exceptions (CONSTRAINT_ERROR, NUMERIC_ERROR, PROGRAM_ERROR, STORAGE_ERROR, and TASKING_ERROR) may be raised explicitly with raise statements and may have handlers written for them.

T2. Check that CONSTRAINT_ERROR, NUMERIC_ERROR, PROGRAM_ERROR, STORAGE_ERROR, and TASKING_ERROR are not reserved words.

Implementation Guideline: Check not only that the names can be redeclared as variables, but also that redeclared names cannot still be used in a raise statement (see also IG 11.2/T1 and IG 11.3/T1).

T3. Check that a user-defined exception having the same name as a predefined exception is distinct from the predefined exception.

AD-A189 647

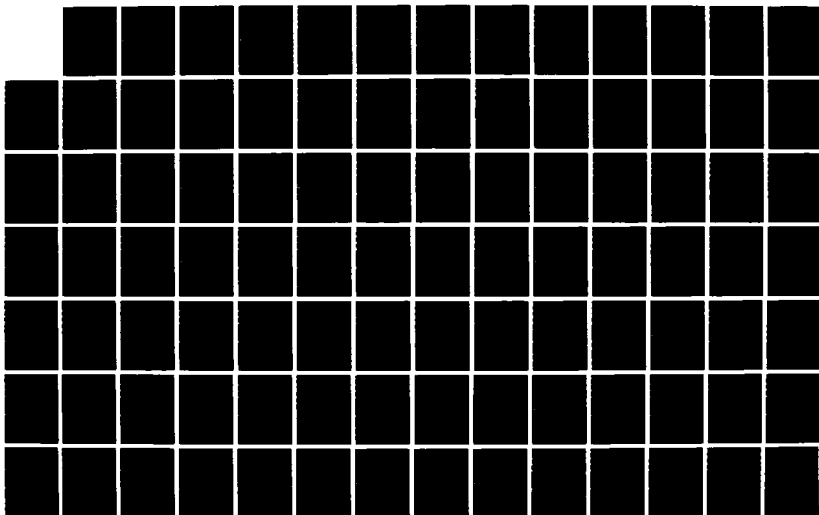
THE ADA (TRADE NAME) COMPILER VALIDATION CAPABILITY
IMPLEMENTERS' GUIDE VERSION 1(U) SOFTECH INC WALTHAM MA
J B GOODENOUGH DEC 86

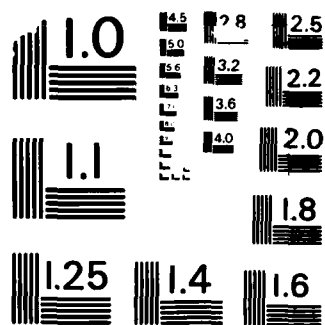
7/9

UNCLASSIFIED

F/G 12/5

NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

- T4. Check that an exception declared in a recursive procedure, unlike a variable declared in a recursive procedure, is not considered different for each recursive activation of the procedure, i.e., different recursive activations should all refer to "the same" exception entity. Check that this holds also for exceptions declared in blocks and packages inside recursively called procedures.
- T5. Check that exceptions declared in generic packages or procedures are considered distinct for each instantiation.
- Check that an exception name declared in a generic package instantiation in a recursive procedure denotes the same entity even when the instantiation is elaborated more than once because of recursive calls.
- T6. Check that `SELECT_ERROR` and `'FAILURE` are no longer predefined exceptions.
- T7. Check that `CONSTRAINT_ERROR` is raised by constraint violations. (This objective is tested wherever constraint violations occur).
- T8. Check that `NUMERIC_ERROR` is raised by predefined numeric operations that cannot deliver a correct result (see IG 4.5.3.a/T4, IG 4.5.3.b/T22, IG 4.5.3.c/T32, IG 4.5.4.a/T12, IG 4.5.4.b/T22, IG 4.5.4.b/T32, IG 4.5.5.a/T4, IG 4.5.5.b/T23, IG 4.5.5.b/T34, IG 4.5.6.a/T13, IG 4.5.6.b/T22-T24, IG 4.5.6.c/T32, IG 4.5.6.d/T42, and IG 4.5.6.d/T52).
- T9. Check that `PROGRAM_ERROR` is raised:
- if execution reaches the end of a function (see IG 6.5/T3);
 - upon an attempt to call a subprogram, activate a task, or elaborate a generic instantiation when the corresponding bodies have not yet been elaborated (see IG 3.9/T6-T8);
 - by the execution of a selective wait that has no else part when all alternatives are closed (see IG 9.7.1/T17).
- T10. To the extent possible and reasonable, check that `STORAGE_ERROR` is raised when:
- storage allocated to a task is exceeded;
 - storage for allocated objects is exhausted;
 - storage for a declarative item is insufficient;
 - storage for execution of a subprogram call is insufficient.

11.2 Exception Handlers

Semantic Ramifications

Changes from July 1982

- S1. There are no significant changes.

Changes from July 1980

- S2. The term "frame" is introduced.

Legality Rules

- L1. Each name occurring as an exception choice must have been declared as an exception.
- L2. A set of exception choices must not refer to a particular exception more than once, either

within a given exception handler or in a sequence of exception handlers within the same frame.

- L3. An **others** exception choice, if present, must appear as the only choice in the last handler of the sequence of handlers.

Test Objectives and Design Guidelines

These exception handlers should be designed so as not to raise any exceptions, either explicitly or implicitly. In particular, the exception being handled is not to be re-raised.

Note: Single-level exception handling for all predefined exceptions will be checked implicitly as a result of tests for constructs raising predefined exceptions. These tests are associated with the RM sections where the constructs are defined. However, an additional test is defined in IG 11.4/T4.

- T1. Check that the name in a handler must be an exception (predefined or programmer-defined). (Use of the notation `STANDARD.NUMERIC_ERROR`, etc. in a handler is tested in IG 8.6/T6. Checks that a declared exception name is or is not hidden appropriately are performed in IG 8.3/T21-T25.)

- T2. Check that an exception cannot be referred to more than once in a single handler or in a sequence of handlers.

Implementation Guideline: Try names of the form `STANDARD.NUMERIC_ERROR`, and `NUMERIC_ERROR`, as well as `P.SINGULAR` and `SINGULAR` (where `P` is a procedure and `SINGULAR` is a locally defined exception). Try names that rename exceptions as well.

- T3. Check that **others** can only appear by itself at the end of a sequence of exception handlers, and more than one **others** choice is not allowed.

- T4. Check that a

- predefined exception, or
- programmer-defined exception,

raised several (at least 3) levels inside a hierarchy of nested blocks, can be successfully handled in an outer block several levels away from the place of occurrence. Handlers containing:

- a single `exception_choice`,
- several `exception_choices`, and
- **others**

should be tried. The handler should not itself raise any exception. One or two other exception handlers, including one mentioning the exception by name, should be present in some block textually enclosing the first exception handling block.

Implementation Guideline: Try nested blocks both inside and outside an exception handler.

- T5. Check that a `return` statement can appear in an exception handler, causing control to leave the subprogram containing the handler.

Implementation Guideline: Check both functions and procedures.

- T6. Check that local variables (and parameters) of a subprogram, a package, or a task are accessible within a handler.

- T7. Check that an `exit` statement in a handler can transfer control out of a loop.

11.3 Raise Statements

Semantic Ramifications

S1. The identity of an exception causing a handler to be executed must be stored so the exception can be further propagated, if necessary, by a raise statement that does not name an explicit exception. The storage place used to hold the identity of the exception must, in general, be allocated within the handler's frame; it is inadequate to allocate a single location per task:

```
when A | B =>
begin
    raise CONSTRAINT_ERROR;
exception
    when CONSTRAINT_ERROR => null;
end;
raise;                -- must raise A or B, not CONSTRAINT_ERROR
```

Changes from July 1982

S2. There are no changes.

Changes from July 1980

S3. There are no significant changes.

Legality Rules

- L1. The innermost construct enclosing a raise statement not mentioning any exception name must be an exception handler, not a subprogram, package, task, or generic unit.
- L2. The name given in a raise statement must denote an exception.

Test Objectives and Design Guidelines

- T1. Check that the identifier mentioned in a raise statement must be an exception (predefined or programmer-defined).

Implementation Guideline: Redecare the predefined exception names as nonexceptions, e.g., as INTEGER, and then try to use the redclared names in a raise statement. In addition, try a programmer-defined integer variable name and undeclared names in an exception handler.

- T2. Check that a raise statement that does not name an exception is forbidden outside an exception handler.

Implementation Guideline: In addition to simple cases, try raise statements in contexts like the following:

```
when E1 | E2 =>
declare
    RANGE_ERROR : exception;
    procedure P is -- use a package, task, and generic unit
    begin
        raise; -- illegal
    end P;
begin
    raise RANGE_ERROR;
exception
    when RANGE_ERROR => P;
end;
```

- T3. Check that a raise statement containing no exception name propagates the exception being handled to another handler; use the raise statement in

- a handler specific to that exception,

- a handler for several exceptions including the one being propagated,
- a handler for others.

Implementation Guideline: Use one case where an exception is raised and handled within the handler before the initial exception is propagated further by raise;.

- T4. Check that when an inner unit redeclares an exception name (either predefined or programmer-defined), thereby hiding the definition previously in effect, then the hidden definition is still available for use by component selection (from the outer module, or, respectively, from the predefined module STANDARD).
- T5. Check that a statement of the form

```
raise [exception_name] when condition;
```

is forbidden.

11.4 Exception Handling

Semantic Ramifications

- S1. Both RM 11.4.1 and RM 11.4.2 are discussed here.

S2. It is possible to propagate an exception out of a compilation unit. When the compilation unit is a library package or a main program, and an exception is propagated, the execution of the main program is abandoned. The RM does not define what it means to abandon execution. In particular, it is implementation-dependent whether the exception is propagated to the external environment. However, to help in debugging programs, such exceptions should be made available to the main program's environment. Moreover, an implementation should preserve the context giving rise to an exception until a handler is found; if no handler exists, then the preserved context can be used by the main program's environment to provide useful information about the situation giving rise to the exception.

- S3. For package body subunits that raise an exception, the exception is propagated to the environment containing the subunit's stub. For example:

```
procedure P is
  package Q is
    ...
  end Q;
  package body Q is separate;
begin ... end P;
```

If an exception is raised during the elaboration of Q's body, the exception will be propagated to P's caller, just as would be the case for any exception raised in P's declarative_part.

S4. Exceptions, unlike variables, can exist outside the scope of their declaration. In particular, they can be propagated into scopes where they cannot be named, and in such cases, they can be handled only by an others handler. Moreover, it is possible to propagate an exception out of its scope and then back into its scope (see below). The practical consequence is that all declared exceptions must be uniquely identifiable at run time independent of the scope in which they are declared. For example, consider the following set of packages and procedures. They have been designed so the exception Q.ECP is propagated out of its scope and then back into its scope. The idea is that Q.G calls P.F, which calls Q.H, which raises exception Q.ECP. Since ECP is local to Q and not known to P, it is propagated out of its scope to F and back into its scope to G.


```

package P is
  procedure F;           -- called by Q.G
end P;

package Q is
  procedure G;           -- call this procedure first
  procedure H;           -- called by P.F
end Q;

with P;
package body Q is
  ECP : exception;
  procedure G is
  begin
    P.F;                 -- call F;
    exception
      when ECP => ...    -- Q.ECP can be handled here, even
                        -- after it has been propagated out
                        -- of the scope of its declaration.

  end G;

  procedure H is
  begin
    raise ECP;           -- propagated to H's caller
  end H;
begin
  Q.G;                   -- start execution
end Q;

with Q;
package body P is
  ECP : exception;       -- distinct from Q.ECP
  procedure F is
  begin
    Q.H;                 -- call H
    raise ECP;           -- raise P.ECP (won't be executed)
    exception
      when ECP => ...    -- only handles P.ECP, not Q.ECP
      when others => raise; -- handles Q.ECP
  end F;
end P;

```

Note that all the above units can be compiled separately or together in a declarative part (without the with clauses) without affecting the propagation and handling of Q.ECP.

Changes from July 1982

S5. There are no significant changes.

Changes from July 1980

S6. The execution of a task is completed rather than terminated when an exception is raised but not handled within the task.

S7. If a frame is a subprogram or a block statement with dependent tasks, then an exception is not propagated until all dependent tasks have terminated.

sa. When a task being activated raises an exception in its declarative part, `TASKING_ERROR` is raised at the point of activation instead of the original exception.

Test Objectives and Design Guidelines

- T1. Check that any exception raised in the sequence of statements of a subprogram body is propagated to callers of the subprogram, not to the statically enclosing lexical environment.

Implementation Guideline: Both predefined and programmer-defined exceptions should be checked.

Implementation Guideline: Some of the subprograms in the dynamic call chain should have exception handlers and some should not. Automatic propagation through more than one level of call should be attempted in some test cases.

- T2. Check that an exception raised during the elaboration of a subprogram's declarative part is propagated to the caller of the subprogram, not to the environment statically enclosing the subprogram's body.

Implementation Guideline: The subprogram body should contain a handler for the exception raised in the declarative part (although this handler should not be executed).

Implementation Guideline: All predefined exceptions can occur when declarative parts are elaborated. The compile-time processing of initializations and the checking of constraints need not be performed by the same compiler code that processes assignment statements. However, we assume that it would be extremely unlikely for a compiler to propagate one exception correctly and not another. Therefore only the following 3 cases need be included in the tests:

- a predefined exception propagated from a function called when elaborating the declarative part;
- a predefined exception raised when elaborating a constraint and by initialization;
- a programmer-defined exception propagated out of a function called when elaborating the declarative part.

- T3. Check that exceptions raised during the elaboration of package specifications or declarative parts of blocks and package bodies are propagated to the environment statically enclosing the block or package (note that no exceptions are propagated from a task body; see RM 11.4.1/9 and IG 11.4/T13).

As a subcase, check specifically that exceptions raised by functions invoked during the elaboration of a declarative part are always propagated outside the declarative part.

Implementation Guideline: The block or package should have a handler for the exception being propagated (although this handler should not be executed).

- T4. Check that

- an implicitly raised exception,
- an explicitly raised predefined exception,
- an explicitly raised programmer-defined exception, and
- a programmer-declared exception having the same name as a predefined exception

occurring in the body of a unit possessing:

- a handler specific to the exception,
- a handler for several exceptions ("A or B or C") including the one that occurred, and
- an others handler

is always handled locally.

- T5. Check that exceptions propagated out of a handler are propagated outside the unit

containing the handler, without invoking some other handler associated with that unit or recursively re-invoking the same handler.

Implementation Guideline: There should be no blocks inside these handlers.

- T6. Check that exceptions occurring in a block declared inside an exception handler follow the rules for blocks.
- T7. Check that the statement part of package bodies can raise, propagate, and handle exceptions. Check also that if a package body's exception handler handles all exceptions raised within the sequence of statements and does not raise an unhandled exception, no exception is propagated out of the package body.
- T8. Check that "systematic unwinding" is possible: an exception occurring deep inside a combined static/dynamic hierarchy is handled successively by each unit in the propagation path, each handler on the path performing its unit's "last wishes" and re-raising the exception to allow the remaining units on the propagation path to perform *their* "last wishes."

Implementation Guideline: Use blocks, packages, and subprograms.

- T9. With a user-defined exception, check that the exception can:
 - be handled out of its scope only with a handler for *others*, then be re-raised for dynamic propagation back into its original scope; and, finally,
 - be handled again, back in scope, under its original name.

Check also that predefined exception names redeclared by the user behave similarly (with the difference that out-of-scope a specific handler for that name refers to the predefined exception).

Implementation Guideline: Check using separately compiled units as well as single units.

- T10. Check that when an exception is raised in a subprogram or block with dependent tasks, no exception is propagated until all the dependent tasks have terminated (see IG 9.4/T1).
Check that an exception is propagated from a package body even if the body declared some task objects that are not yet terminated (see IG 9.4/T7).
- T11. Check that unhandled exceptions raised in package subunits are propagated to the environment statically enclosing the corresponding body stub.
- T12. Check that when an unhandled exception is raised in the main program, the execution of the main program is abandoned.
- T13. Check that exceptions raised in the declarative part of a task body raise TASKING_ERROR at the point of task activation, not the original exception (see IG 9.3/T4).
Check that exceptions raised in task bodies do not propagate outside the task.
- T14. Check that when exceptions are raised during the elaboration of a library unit, execution of the main program is abandoned.

11.5 Exceptions Raised During Task Communication

Semantic Ramifications

Changes from July 1982

- S1. There are no significant changes.

Changes from July 1980

S2. There are no significant changes.

Exception Conditions

- E1. If task S calls an entry in task T, **TASKING_ERROR** is raised in S at the entry call if T is completed at the time of the call or prior to accepting the call.
- E2. During a rendezvous, **TASKING_ERROR** is raised in the calling task if the task containing the accept statement is abnormally terminated as the result of an abort statement.

Test Objectives and Design Guidelines

- T1. If a user-defined or predefined exception is raised during a rendezvous but is not handled within the rendezvous, check that the exception is propagated both within the calling task (at the point of the entry call) and within the called task at the point of the accept statement.
- T2. Check that **TASKING_ERROR** is raised under the appropriate conditions:
 - a. when the called task is completed at the time of the call (**TASKING_ERROR** is raised in the calling task);
 - b. when the called task is not completed at the time of the call, but terminates before the entry call is accepted (**TASKING_ERROR** is raised in the calling task);
 - c. when **TASKING_ERROR** is raised explicitly or by propagation within the accept statement (**TASKING_ERROR** is raised in both tasks);
 - d. when the called task is terminated by an abort statement during rendezvous (**TASKING_ERROR** is raised in the calling task);
- T3. Check that when the calling task is aborted during a rendezvous, no exception is raised in the called task.

Check that when the calling task is aborted before the rendezvous has started, the entry call is cancelled.

11.6 Exceptions and Optimizations**Semantic Ramifications**

S1. The purpose of this section of the RM is to indicate that certain optimizations can be provided by an implementation. One set of optimizations permits evaluating a function earlier than its textual position would indicate, as long as the function would eventually be evaluated before control leaves a certain region of text, namely the statements between *begin (or do)* and *exception* or *end*. In short, a certain amount of imprecision is allowed with respect to when an exception is raised in such a sequence of statements. The rules allow the evaluation of arithmetic operations in parallel or with pipelined arithmetic units even if the existence of **NUMERIC_ERROR** is not indicated until control has entered some statement following an arithmetic expression. The only requirement is that no exception due to evaluating some operation be raised after control has passed out of a frame or into an inner frame. Hence, if a programmer requires more precision over the region of text where a particular exception can be raised, then the desired text can be enclosed in a block.

S2. The rules are written primarily with the predefined operations in mind, but they are stated so that they apply to user-defined functions as well if a compiler can determine that they are functions satisfying the criteria listed in RM 11.6/2.

S3. Since an implementation can suppress the evaluation of operations yielding unneeded values (RM 11.6/7), boolean expressions can always be short-circuit evaluated if no user-defined subprograms are invoked, e.g.,

```
if TRUE or A = B(50) then
```

need not raise `CONSTRAINT_ERROR` even if `B'LAST` is less than 50.

S4. Since the order of operand evaluation is not defined by the language, given

```
begin
  D := 30;
  if A = D/(A-3) or A = B(50) then ...
exception
```

there is no guarantee that `NUMERIC_ERROR` will be raised instead of `CONSTRAINT_ERROR` if `A = 3`. Either operand of `or` can be evaluated first. Moreover, if `CONSTRAINT_ERROR` is raised, then there is no guarantee that `D = 30` in a handler for this exception, since `B(50)` can be evaluated before the assignment to `D` is performed. (The canonical order of evaluation requires assigning 30 to `D` before evaluating `B(50)`, but under the assumption that no predefined operation propagates an exception (RM 11.6/4), the effect is unchanged if `B(50)` is evaluated prior to the assignment to `D`, and so this evaluation order is allowed.) However, given

```
begin
  D := 30;
  if A = D/(A-3) or else A = B(50) then ...
exception
```

`B(50)` cannot be evaluated before `D/(A-3)` has been evaluated (since `B(50)` will not always be evaluated in the canonical order if no exceptions are raised), and hence, if `CONSTRAINT_ERROR` is raised, `D` must equal 30.

Changes from July 1982

S5. There are no significant changes.

Changes from July 1980

S6. The innermost enclosing frame must be the same in the alternate order as in the canonical order, and the same exception handler must apply.

S7. Reordering of operand associations within an expression is allowed even if the reordering removes an exception or introduces a predefined exception.

S8. An implementation can use an operation of a type with a wider range than that of the base type of the operands so long as it delivers the correct final result. Intermediate results may lie outside the range and `NUMERIC_ERROR` need not be raised.

Test Objectives and Design Guidelines

Since an implementation need not provide any of the optimizations permitted by this section, we do not give any test objectives for this section in this version of the IG. Later versions will contain test objectives to check that the permitted optimizations are not applied inappropriately.

11.7 Suppressing Exceptions

Semantic Ramifications

S1. This pragma grants permission to the compiler to suppress certain checks. It does not issue an order to suppress the checks; therefore, the compiler can choose not to suppress the checks.

S2. There are some checks that cannot be suppressed because the exception situation is not covered by the wording for any check name. These situations are:

- checking whether a called task is complete or terminated when an entry call is made (RM 11.5/2);
- checking whether a task is completed abnormally during a rendezvous (RM 11.5/5);
- in an array conversion, checking that the constraint for the operand's component type equals the constraint for the type mark's component type (RM 4.6/13);
- checking that a function is left other than by returning a value or by raising an exception (RM 6.5/2).

S3. The name used as the second argument must be a simple name or an expanded name. Therefore, checks on record or on array components cannot be suppressed.

S4. Note also that names which are not in scope cause the pragma to be ignored (see RM 2.8/9). (However, the compiler should issue a warning.)

S5. Note that `ACCESS_CHECK`, `DISCRIMINANT_CHECK`, `INDEX_CHECK`, `LENGTH_CHECK`, and `RANGE_CHECK` can be applied to any type or object, including types and objects for which these checks do not apply. In general, the intent is to allow the suppression of checks for only the objects and types to which the specified checks apply. An implementation must ignore `SUPPRESS` pragmas that attempt to suppress checks for types that do not apply.

Changes from July 1982

S6. There are no significant changes.

Changes from July 1980

S7. The name in a `SUPPRESS` pragma must be a simple name or an expanded name.

S8. The `SUPPRESS` pragma can appear within a package specification.

S9. The `SUPPRESS` pragma can be applied to names of generic units and names of subprograms.

S10. The effect of applying `SUPPRESS` to a task is defined.

S11. `ACCESS_CHECK` applies now to the use of attributes and to slices.

S12. The range check implicit in checking index values for aggregates and actual parameter values of generic instantiations can now be suppressed.

S13. `DIVISION_CHECK` and `OVERFLOW_CHECK` can only be suppressed for numeric types.

S14. `ELABORATION_CHECK` has been added.

S15. `STORAGE_CHECK` can only be suppressed for access types, task units, and subprograms.

Legality Rules

If any of these conditions is not satisfied, the pragma has no effect (RM 2.8/9).

- L1. The identifier appearing as the first argument of a SUPPRESS pragma must be one of the predefined names.
- L2. The pragma can only appear in a declarative part of a frame or in a package specification.
- L3. ACCESS_CHECK, DISCRIMINANT_CHECK, INDEX_CHECK, LENGTH_CHECK, and RANGE_CHECK can only be applied to an object or a type.
- L4. DIVISION_CHECK and OVERFLOW_CHECK can only be applied to numeric types.
- L5. ELABORATION_CHECK can only be applied to task units, generic units, and subprograms.
- L6. STORAGE_CHECK can only be applied to names denoting access types, task units, and subprograms.

Test Objectives and Design Guidelines

Since a SUPPRESS pragma need not be obeyed, the only tests defined at this time are those needed to check that the pragma is recognized properly. Tests to evaluate whether SUPPRESS is being obeyed will be specified in later versions of the Implementers' Guide. The currently specified tests indicate the minimum support every implementation must provide for SUPPRESS.

- T1. Check that an identifier ending in _CHECK, but not one of the predefined check names, cannot be given as the first argument of SUPPRESS.
Check that SUPPRESS is only permitted in a package specification or in the declarative part of a frame.
- T2. Check that any of the predefined check_names are permitted as the first argument of SUPPRESS, namely, names beginning with ACCESS, DISCRIMINANT, INDEX, LENGTH, RANGE, DIVISION, OVERFLOW, ELABORATION, and STORAGE and ending with _CHECK.
Implementation Guideline: Try a SUPPRESS pragma even when objects and types have been declared with one of these check_names.
- T3. Check that the second argument of SUPPRESS can only be an object name, a type name, or a subtype name (including task type names) when the first argument is ACCESS_CHECK, DISCRIMINANT_CHECK, INDEX_CHECK, LENGTH_CHECK, or RANGE_CHECK.
Implementation Guideline: Include some forms of SUPPRESS with ON.
- T4. Check that the second argument of SUPPRESS can only be the name of a numeric type when the first argument is DIVISION_CHECK or OVERFLOW_CHECK.
- T5. Check that the second argument of SUPPRESS can only be a name denoting a task unit, a generic unit, or a subprogram when the first argument is ELABORATION_CHECK.
- T6. Check that the second argument of SUPPRESS can only be a name denoting an access type, a task unit, or a subprogram when the first argument is STORAGE_CHECK.

Chapter 12

Generic Units

12.1 Generic Declarations

Semantic Ramifications

S1. In this section, we will consider those issues that are unique to generic declarations as a whole. Issues that apply only to certain forms of generic parameters will be discussed in later subsections.

S2. Although generic units have many properties similar to those of macros, a generic declaration cannot be treated simply as a body of unanalyzed text. In particular, the legality of a generic declaration is independent of whether it is ever elaborated, or ever used, in an instantiation.

S3. In accordance with RM 8.3/5, the declaration of the generic unit is not visible either directly or by selection within the unit's generic formal part. In addition, since a generic subprogram is declared using a subprogram specification (preceded by a generic formal part), and since RM 8.3/16 gives a rule applicable to any subprogram specification, a generic subprogram's identifier cannot be used within the subprogram's specification:

```
-- a library unit
generic
  X : INTEGER := F;           -- illegal; RM 8.3/5
  Y : INTEGER := STANDARD.F;  -- illegal; RM 8.3/5
  Z : INTEGER := F.X;         -- illegal; RM 8.3/5
  W : INTEGER := X;           -- legal
function F (
  A : INTEGER := F.W;         -- illegal; RM 8.3/16
  B : INTEGER := W;           -- legal
  return INTEGER;
```

Within F's body, F is visible but it still cannot be used within the subprogram's specification:

```
function F (
  A : INTEGER := F.W;         -- still illegal; RM 8.3/16
  B : INTEGER := W;           -- legal
  return INTEGER is ...
```

RM 8.3/16 forbids the use of any identifier F within a subprogram specification, even if the identifier does not denote the subprogram (see IG 6.1/S for further examples). For example, the identifier F cannot be used as the name of a record component in a component selection in a default expression for one of F's parameters. Since RM 8.3/16 does not apply to a generic formal part, however, such a component selection would be allowed there:

```
type REC is
  record
    F : INTEGER;
  end record;

R : REC;
```



```

generic
  X : INTEGER := R.F;           -- legal; F denotes record component
function F (Y : INTEGER := R.F); -- illegal use of F; RM 8.3/16

```

S4. The identifier of a generic unit hides from the beginning of the declaration's immediate scope (see RM 8.3/14-15), i.e., from the beginning of the generic formal part (see RM 8.2/2). A generic package is not overloadable either inside or outside the unit. Consequently, any use of the package's identifier within the generic formal part is illegal, since all outer entities having the package's identifier are hidden, and the unit's name is not visible until after the end of the generic formal part (RM 8.3/5):

```

function F return INTEGER;

with F;
package P is
  X : INTEGER := F;           -- invokes the library unit

  generic
    Y : INTEGER := F;         -- illegal; no F is visible
    W : INTEGER := STANDARD.F;
  package F is
    Z : INTEGER := F;         -- illegal; only pkg F visible
    V : INTEGER := STANDARD.F; -- ok
  end F;

  Z : INTEGER := F;           -- illegal; only the generic unit is visible
                                -- STANDARD.F would be legal.
end P;

```

The F named in the initialization expression for F.Y cannot denote STANDARD.F because the F denoting STANDARD.F is hidden by the declaration of P.F. STANDARD.F can be used in the initialization of W, however, because generic unit F's declaration only affects direct visibility of STANDARD.F (not visibility by selection).

S5. The identifier of a generic subprogram is overloadable within the generic unit, since within the unit, the identifier is considered to denote a subprogram, not a generic unit (RM 12.1/5). Within the generic formal part, however, the identifier is not considered overloadable, so an implementation need not consider the parameter and return type profile in deciding what declarations are visible:

```

function F return FLOAT;

with F;
package P is
  X : FLOAT := F;           -- invokes the library unit

  generic
    Y : FLOAT := F;         -- illegal; STANDARD.F is hidden
    W : FLOAT := F (3);     -- illegal; P.F not yet visible
  function F (A : INTEGER) return FLOAT;

  Z : FLOAT := F;           -- illegal; only the generic unit is visible
                                -- STANDARD.F would be legal.
end P;

```

Within F's generic formal part, P.F is not considered overloadable, and so hides STANDARD.F. P.F is also not visible within the formal part, so the initialization of Y is illegal. Similarly, P.F would be illegal as the initialization expression for W. STANDARD.F would be allowed as the initialization expression for Y, since it is only the declaration of P.F that is not visible in the generic formal part; other declarations of F are visible by selection. Within F's body, Y := F would be legal and so would W := F(3), since within the body, P.F is overloadable.

s6. Although the identifier of a generic unit hides before it textually appears, a formal subprogram parameter is visible after its declaration:

```
function F return INTEGER;

with F;
package P is
  generic
    Y : INTEGER := F;           -- illegal; STANDARD.F is hidden
    with function F (X : INTEGER := 1) return INTEGER;
    W : INTEGER := F;           -- unambiguous; P.F.F
    Z : INTEGER := F (3);       -- legal; P.F.F
  function F (A : INTEGER := 0) return INTEGER;
end P;
```

The initialization of W is unambiguous because STANDARD.F is hidden and P.F is not yet visible. The only visible F is the formal subprogram, P.F.F. (Note: the expanded name, P.F.F, could not actually be written as W's initialization expression since P.F is not yet visible.)

s7. Here are some more examples showing how formal parameter declarations follow the usual visibility rules:

```
package P is
  type T is (A, B, C);
  R : INTEGER;

  generic
    Q : INTEGER := T'SIZE;      -- size of STANDARD.P.T
    R : INTEGER := P.Q'SIZE;    -- illegal; no P is
                                -- directly visible
    S : INTEGER := P.T'SIZE;    -- illegal; no P is
                                -- directly visible

    type T is range <>;
    U : INTEGER := T'SIZE;      -- size of P.P.T when
                                -- P.P is instantiated
    U1 : INTEGER := STANDARD.P.P.T'SIZE; -- illegal (see below)
    V : INTEGER := P.R'SIZE;    -- illegal; no P is
                                -- directly visible
    V1 : INTEGER := STANDARD.P.R'SIZE; -- ok (RM 8.3/13)
  package P is
    ...
  end P;
end P;
```

Within the generic formal part of P, the expanded name STANDARD.P.P is illegal because generic unit P is neither directly nor selectively visible within its own generic formal part (see RM 8.3/5). Hence, STANDARD.P.P.T'SIZE is illegal.

s8. To summarize the effect of the visibility rules for generic formal parts:

- RM 8.3/5 implies a generic unit is not visible either directly or by selection within its generic formal part. Hence no expanded name (e.g., STANDARD.P.P) is allowed whose prefix (i.e., STANDARD.P) denotes a construct immediately enclosing the generic unit and whose selector (i.e., the final P in STANDARD.P.P) is the generic unit's identifier. The identifier of the generic unit is allowed to denote other entities by selection, however, and so it can be used:
 - as the selector in an expanded name (e.g., STANDARD.P) whose prefix (i.e., STANDARD) does not denote a construct immediately enclosing the generic unit,
 - as the selector denoting a record component (or a discriminant) (e.g., REC.P),
 - as a choice in a record aggregate (e.g., (P => 5), and
 - as a formal parameter name in a named association of a function call (e.g., F (P => 5)).

(These are the only selection contexts applicable in generic parts; see RM 8.3/7-13.)

- The identifier of a generic unit is not considered overloadable within its generic formal part, and so hides all outer declarations with the same identifier. Hence, if no formal parameter has the same identifier as that of the generic unit, then the identifier of a generic unit is not allowed in a generic formal part as a type mark, a primary, or a function name, nor is it allowed as the prefix of an indexed component, a slice, a selected component, or an attribute, or as the default name in a formal subprogram declaration.

s9. Names within the body of a generic unit are bound in the context of the body, just as for nongeneric subprograms and packages. In particular, identifiers declared in a generic package or subprogram declaration are directly visible within the corresponding body (see IG 12.2/S). Since the body of a generic package or subprogram is textually separate from its declaration, different identifiers can be visible in the different contexts, and so different bindings can occur:

```
generic
  type T is range <>;
  X : INTEGER;
package P is
  procedure R (Z : T);
end P;

X : INTEGER;
procedure R (Y : INTEGER);

package body P is
  procedure R (Z : T) is ... end R;
begin
  R (X);    -- equivalent to R (Y => P.X)
  R (0);    -- ambiguous; R (Z => 0) or R (Y => 0)
end P;
```

The first invocation of R is equivalent to R(P.X) because the declaration of X as a formal generic parameter hides the outer declaration of X. The second call is ambiguous, because 0 could have type T or type INTEGER, and there are overloads of R having parameters of type T and INTEGER. (See also IG 8.3.f/S.)

S10. Within a generic subprogram or package, the generic unit has the usual properties of a subprogram or package. In particular, entities declared within a generic subprogram or package can be named using component selection, starting with the name of the generic unit. In addition, for generic subprograms, calls naming the generic subprogram can be written within the unit; subprograms that overload the unit's identifier can be declared inside the unit; and the name of the unit can appear as an actual parameter in a generic instantiation:

```
generic
  type U is range <>;
  with function F (X : U) return U;
package P is
  X : INTEGER := 0;
  Y : INTEGER := P.X;           -- legal
end P;

Z : INTEGER := P.X;           -- illegal; P.X not visible

generic
  type T is range <>;
  function GEN_FACT (X : INTEGER) return INTEGER; -- overloads unit
function GEN_FACT (X : T) return T;

function GEN_FACT (X : T) return T is
  procedure GEN_FACT (X : INTEGER) is ... end; -- overloads unit
                                              -- and formal parameter
  package NP is new P (T, GEN_FACT); -- means instance of GEN_FACT
begin
  if X > 1 then
    return GEN_FACT (X-1); -- recursion; invokes an instance
                          -- of STANDARD.GEN_FACT
  else
    return 1;
  end if;
end GEN_FACT;
```

S11. Within a generic unit, the name of the unit denotes an instantiation of the unit, i.e., the name is not considered to denote a generic unit, and so the unit cannot be instantiated within itself:

```
generic
package P is
  package NP is new P; -- illegal; P not generic
end P;
```

Direct and indirect recursive instantiations are also explicitly forbidden by RM 12.3/18 (see also IG 12.3/S).

S12. The name of a generic subprogram is considered overloadable within the unit, but not outside the unit:

```
procedure EX is
```

```

procedure P (X : FLOAT);

procedure P (Y, Z : INTEGER := 4) is

    generic
    procedure P (W : INTEGER := 5);

    procedure P (W : INTEGER := 5) is
    begin
        P (3);           -- ambiguous
        P (W => 3);       -- legal
        P (Y => 3);       -- legal; P (3,4)
        P (3.0);         -- legal; EX.P is overloaded
    end P;               -- EX.P.P

    function P ... is ... end; -- illegal; P not overloadable
                                -- here

begin
    P (4.0);              -- illegal; EX.P hidden
end P;                   -- EX.P

```

These examples illustrate the different interpretations of a generic subprogram's name inside and outside the unit. Inside the body, the generic identifier is overloadable. Since P's type profile is not the same as the type profiles of either EX.P subprogram, the EX.P declarations are not hidden within EX.P.P's body. Hence, EX.P.P overloads the EX.P declarations, and so, P(3) is either EX.P.P (W => 3) or EX.P (Y => 3). Similarly, P(3.0) is equivalent to EX.P (X => 3.0).

Outside EX.P.P, P is not overloadable. Hence, the attempted declaration of function P is illegal. Finally, the call P(4.0) is illegal because the only visible P within the body of EX.P is the generic unit, EX.P.P, and a call to such a unit is forbidden from outside the unit.

The P(3) call would be legal if the generic subprogram had two formal parameters of type INTEGER (with default values). In this case, EX.P.P and the second EX.P would have the same type profile. Hence, the inner P would hide the second EX.P. Hence P(3) could only be interpreted as referring to either the innermost P or to the EX.P that has a FLOAT parameter. Since 3 is not implicitly convertible to FLOAT, the call would be resolved to the inner P.

S13. For a discussion of the situations in which a generic unit can be instantiated before its declaration or body is completely elaborated, see IG 3.9/S. In particular, an instantiation can be written prior to the textual occurrence of the corresponding body, and such an occurrence need not raise an exception at run time. Examples are given later in IG 12.3.2/S.

S14. RM 7.4.4/4 imposes a restriction on the type of an out formal parameter declared "in an explicit subprogram declaration." Subprogram_declaration is a syntactic term. In particular, the specification of a generic formal subprogram is not a subprogram_declaration; such a subprogram can have an out parameter of any limited type. The restriction is explicitly extended to include generic procedure declarations:

```

generic
    type T is limited private;
    type A is array (INTEGER range <>) of T;
    with procedure P (X : out T; Y : out A);      -- ok
    procedure Q (X : out T; Y : out A);          -- illegal

```

Neither the declaration of parameter Q.X nor Q.Y satisfy the requirements of RM 7.4.4/4.

Changes from July 1982

S15. There are no significant changes.

Changes from July 1980

S16. Formal object parameters must be declared with type marks (instead of subtype indications).

S17. Default expressions of formal object parameters are not evaluated unless the value is needed by an instantiation.

S18. It is clarified that within a generic unit, the name is considered to denote a nongeneric unit. In particular, for generic subprograms, the unit's name is considered overloadable and can be used in calls.

Legality Rules

- L1. The designator of a generic subprogram must not be an operator symbol (RM 12.1/4).
- L2. Every subtype indication appearing in a formal array or an access type definition must have the form of a type mark (RM 12.1/4).
- L3. Outside of a generic subprogram body (RM 12.1/5), the name of the generic subprogram must not be used in a subprogram call.
- L4. Outside of a generic package (RM 12.1/5), the name of the generic package must not be used in a use clause (see RM 8.4/1) or as the prefix of an expanded name (see RM 4.1.3/14-15). (Note: RM 4.1.3/16-18 does apply within a generic package.)
- L5. The identifier denoting an object or a type declared as a generic formal parameter must be distinct from all other identifiers declared in the same generic formal part. In addition, for generic subprograms, the identifier must be distinct from all identifiers declared in the subprogram's formal and declarative parts (see RM 8.3/17 and RM 8.1); for generic packages, the identifier must be distinct from all identifiers declared in the package's specification and in the body's declarative part (see RM 8.3/17).

For a generic subprogram, a formal subprogram parameter must not have the same parameter and result type profile (see RM 6.6/1) as another subprogram declared explicitly (RM 8.3/17) in the same generic formal part or in the declarative part of the generic unit's body.

For a generic package, a formal subprogram parameter must not have the same parameter and result type profile as another subprogram declared explicitly (RM 8.3/17) in the same generic formal part, in the package specification, or in the declarative part of the package body.

- L6. The declaration of a generic unit is not directly visible within the unit's generic formal part (RM 8.3/5), nor is the generic unit itself visible by selection within its generic formal part. Consequently, if there is no formal parameter having the same identifier as the generic unit, then the generic unit's identifier is not allowed within the unit's generic formal part as a type mark; a primary; a function name in a function call; a prefix of an indexed component, slice, selected component, or attribute; a default name in a formal subprogram declaration; or a selector in an expanded name whose prefix denotes a construct immediately enclosing the generic unit.
- L7. The identifier of a generic unit must be distinct from all other identifiers declared locally (RM 8.1/8) in the region containing the generic unit's declaration (RM 8.3/17).

In addition, the usual rules for subprogram and package declarations apply to the specification of a generic subprogram and package:

- L8. A default expression is allowed only for formal parameters with mode In (RM 6.1/4).
- L9. The base type of a default expression must be the same as the base type of its formal parameter (RM 6.1/4).
- L10. A simple name is not allowed in a parameter declaration if the name refers to a formal parameter declared earlier in the same formal part (RM 6.1/5).
- L11. The simple name of a generic subprogram cannot be used within the subprogram's formal part except to declare a formal parameter having the name of the subprogram (RM 8.3/16). In particular, its use as a selector in a component selection, as a component simple name in an aggregate, as a parameter name in a named parameter association, or as a simple name in a default expression is forbidden.
- L12. A generic function must only have parameters of mode In (RM 6.5/1).
- L13. An out parameter of a generic subprogram must not have a limited type unless (RM 7.4.4/4):
 - the type is a limited private type,
 - the declaration of the generic subprogram occurs within the visible part of the package that declares the limited private type (including within any nested packages), and
 - the full declaration of the limited private type does not declare a limited type.
- L14. If a generic subprogram declaration is given in a declarative part, then a body (or body stub) must be provided later (RM 12.2/2) in the same declarative part (RM 3.9/9).
- L15. If an identifier is present at the end of a generic package specification, it must be the same as the package identifier (RM 7.1/3).

Test Objectives and Design Guidelines

Implementation Guideline: To check that a generic declaration has a certain property, it may be necessary to instantiate it, or even to invoke the instantiated unit.

- T1. Check that a generic formal object or subprogram declaration cannot contain a reference to itself in its initialization expression or its default name.
- T2. Check that generic In parameters and attributes of generic object parameters are not considered static.

Implementation Guideline: Try using parameters and attributes in choices, accuracy constraints, integer type definitions, fixed point type definitions, representation specifications, or as discriminant values in an aggregate when a variant depends on the value. Include the use of attributes in generic formal parts.
- T3. Check that the only allowed form of subtype indication in a generic formal part is a type mark (see IG 12.1.1/T9 and IG 12.1.2/T1).
- T4. Check that elaboration of a generic package declaration does not elaborate the specification portion of the package.
- T5. Check that the identifier of a generic unit cannot be used in its generic formal part as a type mark, a primary in an expression, a function name in a function call, a prefix of an indexed component, slice, selected component, or attribute, a default name in a formal subprogram declaration, and a selector of an expanded name whose prefix denotes a unit immediately enclosing the generic declaration.

Implementation Guideline: For a generic function, include cases to show that the function is not considered overloadable within its generic formal part. In particular, check for declarations that are potentially visible because of use clauses as well as for declarations given in an outer declarative region.

Check that the generic unit's identifier can be used in its formal part as the selector in an expanded name to denote an entity in the visible part of a package, or to denote an entity immediately enclosed in a construct other than the construct immediately enclosing the generic unit.

Check that the generic unit's identifier can be used in its formal part as a selector to denote a component of a record object, as the name of a record or discriminant component in a record aggregate, and as the name of a formal parameter in a function call.

Check that if a formal subprogram parameter has the same identifier as the generic unit, the identifier can be used in a later formal parameter declaration either as a default subprogram name or in an initialization expression.

T6. Check that a generic formal part cannot precede a subprogram body (see IG 12.2/T1).

T7. Check that a generic unit cannot be instantiated inside the unit itself (see IG 12.3/T9).

If the generic unit is a subprogram, check that it can be named as a generic actual parameter in an instantiation inside the unit (see T15).

T8. Check that the sequence of generic formal parameters in a generic formal part cannot be enclosed in parentheses.

Check that the generic formal parameters in a generic formal part cannot be separated by commas (instead of semicolons).

Check that the last generic formal parameter in a generic formal part must end with a semicolon.

T9. Check that formal object or type parameters cannot have the same identifier.

Implementation Guideline: Include a case where two types have the same identifier but different discriminant specifications.

Check that the identifiers of formal object or type parameters must be distinct from identifiers declared in the specification and body (two cases) of a generic package.

Implementation Guideline: Include cases where the body is given as a subunit.

Check that formal object and type parameters must be distinct from identifiers declared in the formal and declarative parts of a generic subprogram.

Check that generic formal subprograms cannot have the same parameter and result type profile. Use two formal subprogram declarations that are identical except for one of the following differences:

- the parameters are named differently (differences in parameter names are ignored).
- the subtypes of a parameter are different (differences in subtype names are ignored if the base types are the same).
- the result subtypes of two functions are different (differences in subtype names are ignored if the base types are the same).
- the parameter modes are different; also try reordering the parameters and changing their modes.
- a default expression is present/absent (the presence or absence of a default expression does not affect the parameter profile).

Implementation Guideline: See also IG 12.1.3/T8 for legal overloads.

Check that a formal subprogram cannot have the same profile as a subprogram declared later in a package specification or in a generic unit body, using the above cases.

Check that a function declaration equivalent to an enumeration literal is not allowed.

Check that a formal subprogram cannot have the same identifier as a variable, type, subtype, constant, number, array, package, or generic subprogram declared previously in the same declarative region.

- T10. Check that the names in a generic subprogram and package declaration (including its body) are statically identified (i.e., bound) at the point where the generic declaration or body textually occurs, and are not bound at the point of instantiation.

Implementation Guideline: Include a test where a generic declaration and its corresponding body contain the same free identifiers, but are in different contexts, and thus have different bindings for the free identifiers.

Implementation Guideline: Binding of default subprogram names is checked in IG 12.1.3/T1.

- T11. Check that a generic subprogram (i.e., the template) cannot be used as the called subprogram in a subprogram call outside the unit (i.e., an instantiation should be used).

Implementation Guideline: Instantiate the generic subprogram first, but then call the template instead of the instantiated instance. Don't have any generic formal parameters in the generic formal part. Have the call's actual parameters match the nongeneric formal parameters of the generic subprogram's specification.

Check that a generic subprogram cannot be used as an actual parameter corresponding to a formal subprogram parameter if the instantiation is outside the generic unit.

- T12. Check that the name of a generic package (i.e., the template) cannot be used outside the generic package in use clauses, nor as a prefix in an expanded name (i.e., the name of an instantiation should be used).

Implementation Guideline: Instantiate the generic package first, but then use the template name instead of the instantiated name, first in an expanded name, and then in a use clause. Don't have any parameters in the generic formal part.

- T13. Check that generic function designators cannot be operator symbols.

- T14. Check that a generic subprogram cannot be overloaded outside the unit.

Implementation Guideline: Try giving a subprogram with additional parameters.

- T15. Check that the use of a generic identifier is permitted within a generic package or subprogram, and represents the current instance of the generic when it is instantiated.

Implementation Guideline: Check subprograms in calls, and check that selected component notation using the name of the generic unit refers to the current instantiation of the unit. Also, use the name of a generic subprogram as an actual parameter in an instantiation.

- T16. Check that a task or a task type cannot be a generic unit, i.e., these forms of declaration cannot be preceded by a generic formal part.

- T17. Check that a generic formal parameter cannot be a subtype declaration, a derived a type declaration, a record type declaration, or an exception declaration (see IG 12.1.2/T2).

- T18. Check that a formal parameter of a generic formal subprogram can have an out parameter of a formal limited type.

Implementation Guideline: Use both a limited private formal type and an array type.

12.1.1 Generic Formal Objects

Semantic Ramifications

- S1. Although formal parameters of subprograms and generic units both have modes in and in out, there are several important differences between these kinds of parameters:

- for generic units, an in formal parameter is always a copy of the actual parameter's value. For subprogram in formal parameters, the actual parameter may be passed by reference (RM 6.2/7) if it has a composite type.

- In parameters of limited types are forbidden for generic units, but are permitted for subprograms.
- the value of an actual generic parameter is not checked against the constraints of a formal generic in out parameter, but such checks are performed for subprogram parameters.
- the subtype of a generic formal in out parameter is the base type of the type mark used in the parameter's declaration, whereas the subtype of a subprogram formal parameter is that denoted by the type mark. Consequently:

```
subtype TEN is INTEGER range 1..10;
```

```
generic
```

```
  GEN_TEN : in out TEN;
```

```
procedure P (SUB_TEN : in out TEN);
```

```
procedure P (SUB_TEN : in out TEN) is
begin
```

```
  case GEN_TEN is
```

```
    when 1..10 => ...
```

```
    when others => null;      -- required; see RM 5.4/4-5
```

```
  end case;
```

```
  case SUB_TEN is
```

```
    when 1..10 => ...
```

```
  end case;                  -- no others required
```

```
end;
```

- an in out generic formal parameter always denotes (i.e., renames) its actual parameter, whereas a subprogram in out formal parameter having a scalar or access type always denotes a local copy initialized with the actual parameter's value, and may denote a local copy for other classes of type.
- It is not erroneous for different generic formal in out parameters (of the same nonscalar or nonaccess type) to be associated with the same actual parameter; such an association can be erroneous for a subprogram:

```
generic
```

```
  X, Y : in out STRING;
```

```
procedure P;
```

```
procedure P is
```

```
begin
```

```
  X := "AB";
```

```
  Y := Y(1) & Y(1);
```

```
end P;
```

```
R, S : STRING (1..2) := "CD";
```

```
procedure PP (X, Y : in out STRING) is
```

```
begin
```

```
  X := "AB";
```

```
  Y := Y(1) & Y(1);
```

```
end PP;
```

procedure Q is new P (R, R);

An invocation such as PP (R, R) is erroneous, but Q's instantiation is not erroneous (after Q is invoked, R will equal "AA").

It is also not erroneous for an In and an In out generic formal parameter to be associated with the same actual parameter.

S2. If an implementation chooses to generate code for generic units prior to processing any instantiations, it can usually, but not always, pass generic In out parameters by reference. Pass by reference is not convenient, however, when the actual parameter does not lie on an addressable boundary of the target machine, as, for example, when a component of a packed BOOLEAN array or a component of a packed record type is used as an actual In out parameter.

S3. The constraints of a generic formal In parameter are those of the formal parameter when the formal parameter is constrained (RM 12.1.1/3) or has an access type. Otherwise (i.e., for unconstrained formal In parameters having an array type or having a type with discriminants and for all In out formal parameters, constrained or not), any constraints associated with a formal parameter are those of the *actual* parameter. In particular, note the difference between the semantics of constrained In out subprogram formal parameters and In out generic formal parameters — for subprograms, the constraints of the formal parameter are used during the subprogram's invocation; for generic units, the constraints of the formal parameter are ignored; the actual parameter's constraints, if any, are used instead. It is particularly important to keep this difference in mind if an implementation attempts to generate code for a generic body prior to processing any instantiations.

S4. The consequences of these rules are illustrated by the following examples:

```

subtype ONE_FIVE is INTEGER range 1..5;
UP_10 : INTEGER range 1..10;
UP_3  : INTEGER range 1..3;

generic
  INPUT  : ONE_FIVE;
  OUTPUT : in out ONE_FIVE;
procedure P;

procedure body P is
begin
  OUTPUT := INPUT;      -- CONSTRAINT_ERROR possible
end P;
```

The assignment to OUTPUT may raise CONSTRAINT_ERROR, even though INPUT and OUTPUT are declared with the same subtype names. For example, the following instantiation:

```
procedure P1 is new P (4, UP_3);
```

will *not* raise CONSTRAINT_ERROR even though UP_3 and OUTPUT have different constraints. But when P1 is invoked, the value 4 will be checked against the constraints of OUTPUT's actual parameter, UP_3. Since 4 exceeds UP_3's upper bound, the assignment (and P1) will raise CONSTRAINT_ERROR. Now consider:

```
procedure P2 is new P (UP_10, UP_3);
```

This instantiation will raise CONSTRAINT_ERROR only if the value of UP_10 exceeds 5, the upper bound of the formal In parameter. If the instantiation succeeds, then the invocation of P2 will raise CONSTRAINT_ERROR if the value of UP_10 exceeds 3, the upper bound of UP_3.

S5. Now consider declarations with array types:

```

subtype STR3 is STRING (1..3);
STR : STRING (1..10) := "9876543210";
generic
    IN_S3      : STR3;
    IN_SU      : STRING;
    INOUT_S3   : in out STR3;
    INOUT_SU   : in out STRING;
procedure Q;

procedure body Q is
    V1 : CHARACTER := IN_S3(3);
    V2 : CHARACTER := IN_SU(3);
    V3 : CHARACTER := INOUT_S3(3);
    V4 : CHARACTER := INOUT_SU(3);
begin
    INOUT_S3 := "ABCD";
    INOUT_SU := "DEF";
end;

```

Since IN_S3 is declared as a constrained array and is an In parameter, it has the bounds of STR3 for each instantiation:

```

procedure Q1 is new Q (IN_S3 => STR (2..4),
    IN_SU => STR (2..4);
    INOUT_S3 => STR (2..5),
    INOUT_SU => STR (3..5));

```

Hence for Q1, IN_S3 has the value "876", IN_S3'LAST = 3, and V1 = '6'. On the other hand, since IN_SU is unconstrained, the bounds of IN_SU are determined by the bounds of the actual parameter in each instantiation. (The actual parameter value is assigned to the formal parameter as for a constant declaration; see RM 12.3/7). Consequently, Q1.IN_SU'LAST has the value 4, and Q1.V2 is assigned the value '7'.

The bounds of the formal In out parameters are always those of the corresponding actual parameters (RM 12.1.1/4). Hence, within the instantiation of Q1, INOUT_S3'LAST has the value 5, as does INOUT_SU'LAST. V3 equals '7', and V4 = '7'. After the assignment to INOUT_S3, STR has the value "9ABCD4321", and after the assignment to INOUT_SU, STR has the value "9ADEF4321".

S6. If a formal In parameter has discriminants, the value of its 'CONSTRAINED' attribute is TRUE (RM 3.7.4/3). The value of 'CONSTRAINED' for a formal In out parameter, however, depends solely on the actual parameter.

S7. The name of a formal In out parameter cannot be used in a static expression because In out object parameters are not constants. Similarly, the name of a formal In parameter is not allowed because such parameters are not constants explicitly declared by a constant declaration (RM 4.9/6). Attributes of such parameters cannot be used in static expressions because no attribute of an object is allowed in a static expression (RM 4.9/8).

Changes from July 1982

S8. The subtype of a generic formal object of mode In is the subtype of the formal parameter, not the subtype of the actual parameter.

Changes from July 1980

S9. An in out formal parameter can have a limited type.

Legality Rules

- L1. A default expression for a generic formal in parameter must have the parameter's base type (RM 12.1.1/2).
- L2. Formal in out parameter declarations must not have explicitly specified default initializations (RM 12.1.1/2).
- L3. A generic formal in parameter must not have a limited type (RM 12.1.1/3).
- L4. A generic formal in parameter must not be used as an actual in out or out subprogram parameter, or as the target of an assignment, or as an actual generic in out parameter (RM 12.1.1/3, RM 6.4.1/3, RM 5.2/1, and RM 12.3.1/2).
- L5. The identifier of a formal in parameter is not directly visible in its default expression (RM 8.3/5), and so cannot be used as a primary, as a function name in a function call, or as a prefix of an indexed component, slice, selected component, or attribute.
- L6. No generic object parameter or attribute of a generic object parameter can be used in a static expression (RM 4.9/6, /8).

Test Objectives and Design Guidelines

- T1. Check that object parameter declarations in a generic formal part cannot have the mode out.
- T2. Check that generic formal in out parameter declarations cannot have initializations.
- T3. Check that a generic formal in parameter cannot be used as a subprogram's actual out parameter, or as an actual (generic or subprogram) in out parameter, or as the target of an assignment.
Implementation Guideline: Use a scalar, an array, a record, and an access type as in parameters. For the array and record types, try using their components as well as the entire objects in the forbidden contexts.
- T4. Check that generic formal in parameter declarations cannot have a limited type (in particular, of a limited private, task, or composite type containing such component types).
Check that an in out formal parameter can have a limited type.
- T5. Check that a generic formal in parameter is a copy of the actual parameter value (see IG 12.3.1/T20).
Check that a generic formal in out parameter denotes the actual parameter variable even when the actual variable is not readily passed by reference (see IG 12.3.1/T6).
- T6. Check that mode in is assumed when no mode is explicitly specified in a parameter declaration in a generic formal part.
- T7. Check that a default expression for a generic formal parameter must not refer to a later parameter (neither a type nor an object) of the same generic formal part, nor can the default expression refer to the current parameter.
Implementation Guideline: Use the name of the current parameter as a primary, as the function name in a function call, or as a prefix of an indexed component, slice, selected component, and attribute.
Check that a default expression may refer to an earlier formal parameter of the same generic formal part.
- T8. Check that no attribute of a generic in or in out parameter can be used in a static expression (see IG 4.9/T7).

Check that a generic In out formal parameter does not have a static subtype in a case statement (see IG 5.4.b/T2).

Check that a generic In formal parameter declared with a static subtype is considered to have a static subtype in a case statement (see IG 5.4.b/T2).

Check that a generic In parameter cannot be used in a static expression, even if its subtype is static (see IG 4.9/T7).

T9. Check that a range constraint, an accuracy constraint, an index constraint, or discriminant constraint is not allowed in a formal object declaration.

T10. Check that the default expression for a parameter of mode In must have the same base type as the parameter.

Check that the default expression is not evaluated when the declaration is elaborated (but only when the default value is needed in an instantiation) (see IG 12.3.1/T23).

T11. Check that after a generic unit is instantiated, the subtype of an In out object parameter is determined by the actual parameter.

Implementation Guideline: Use scalar, access, array, record, and private types. Use both constrained and unconstrained formal objects. Use slices as actual parameters. Try X'FIRST when X is a formal parameter with a static lower bound that is different from the bound of the actual parameter.

12.1.2 Generic Formal Types

Semantic Ramifications

S1. The restriction on the form of discrete ranges in array type declarations (RM 12.1.2/2) implies that the following declaration is illegal:

```
generic
  type ARR is array (1..5) of INTEGER;      -- illegal
```

The range 1..5 is not a type mark. In addition, the RANGE attribute is not allowed:

```
generic
  type T is range <>;
  type AR is array (T) of INTEGER;          -- legal
  type AS is array (AR'RANGE) of INTEGER;   -- illegal
```

S2. For purposes of determining the legality of a generic template, any private type declared with discriminants is assumed to have no default discriminant values (RM 12.1.2/3):

```
generic
  type T (D : INTEGER) is private;
package P is
  X : T;      -- illegal
end P;
```

S3. The declaration of a type implicitly declares (RM 12.1.2/13) the predefined and basic operations associated with the type's class. These operations are available for use immediately after the type's declaration, but the implicit operator declarations can be replaced with explicit declarations:

```

generic
  type T is range <>;
  X : T := 5;
  Y : T := X - 3;      -- predefined "-"
  with function "-" (X, Y : T) return T;
  Z : T := Y - X;      -- redefined "-"
package P is
  W : T := Y - Z;      -- redefined "-"
end P;

```

The explicit declaration of "-" is allowed by RM 8.3/17. Note that if T's actual parameter is LONG_INTEGER, the "-" operation in Y's initialization will be the operation predefined for LONG_INTEGER; if T's actual parameter is INTEGER, a different predefined operation will be used.

s4. The implicitly declared operations depend only on the nature of the formal parameter:

```

generic
  type T is (<>);
  X : T;
  Y : T := abs (X);    -- illegal

```

The use of abs is illegal since abs is not defined for both enumeration and integer types (RM 12.1.2/9).

s5. Inside a template, all the formal types of the template are considered distinct from any types declared outside the template (RM 12.1.2/1):

```

X : INTEGER := 0;

generic
  type T is range <>;
package P is
  Y : T := X;          -- illegal; X and Y have different types

```

The situation is different once a template is instantiated:

```

X : INTEGER := 0;

generic
  type T is range <>;
package P is
  subtype ST is T;
  Y : T := 5;
end P;

package P1 is new P (INTEGER);

Z : INTEGER := P1.Y - X;

```

As a result of the instantiation, P1.T denotes the actual parameter, INTEGER (RM 12.3/9). Hence, P1.Y is of type INTEGER, and the expression P1.Y - X is legal.

s6. The following example illustrates some of the finer points concerning what operations are associated with formal types:

```

generic
    type T is limited private;

package LP is
    subtype ST is T;
    type NT is new ST;
end LP;

package LP1 is new LP (INTEGER);
use LP1;

type NST is new LP1.ST;
type NNT is new LP1.NT;

X_ST, Y_ST : LP1.ST;      -- :=, =, 3
X_NT, Y_NT : LP1.NT;      --
X_NNT, Y_NNT : NNT;        --
X_NST, Y_NST : NST;        -- :=, =, 3

```

For which of these types is assignment, equality, and literal notation available? The answer for the derived types depends on the class of the parent type (RM 3.4/5). LP1.ST is an integer type, since LP1.ST denotes T's actual parameter, INTEGER. Therefore, NST has all the operations provided for integer types. In particular, a conversion operation is declared for converting *universal_integer* values to NST values, so numeric literals can be used in NST expressions. Similarly, an equality operator is implicitly declared for NST.

LP1.NT is a limited type because no assignment or equality operations are declared for NT in the template, and so none are declared for NT in the instantiation of the template. (An instantiation is merely a copy of the template with formal parameters bound to actuals in accordance with RM 12.3; in particular, the elaboration of a derived type declaration in an instantiated unit does not cause new implicit declarations to be generated.) Since NNT's parent type is a limited type, so is NNT, i.e., neither assignment, equality, nor literal notation is available for NNT.

One might be tempted to say that since LP1.ST denotes STANDARD.INTEGER, and since LP1.NT is derived from LP1.ST, LP1.NT is an integer type. However, the class of LP1.NT is determined by the operations declared for LP1.NT, and as we have noted, the only operations declared for LP1.NT are those appropriate for a limited type, because these are the only operations declared in template LP. Consequently, LP1.NT does not have an assignment or a predefined equality operation, and is therefore a limited type.

S7. Among the operations declared for LP.NT are operations for converting from LP.NT to its parent type, and vice versa (RM 3.4/5). Consequently, one can write:

```
X : NNT := NNT (LP1.NT (3));
```

The parent type of LP1.NT is LP1.ST (i.e., INTEGER), and an implicit conversion exists from *universal_integer* to LP1.ST. The operation for converting to LP1.NT is implicitly declared in the template LP, and so is available in the instantiation LP1. Finally, conversion from LP1.NT values to NNT values is implicitly declared for NNT. Note that NNT(3) would be illegal, since there is no operation for converting *universal_integer* values to the type NNT.

S8. Now consider:


```

generic
  type D is (<>);
package DIS is
  type AR is array (1..5) of D;
end DIS;

```

Since AR is an array of a discrete type, D, catenation and the relational operators are implicitly declared for AR (RM 3.6.2/12). Since D is not a boolean type, the logical operators and unary not are not implicitly declared. Now consider the following instantiation:

```

generic
package OUTER is
  package DIS1 is new DIS (BOOLEAN); use DIS1;
  X : AR := ...;
  Y : BOOLEAN := X > X;
  Z : BOOLEAN := X and X;  -- illegal
end OUTER;

```

The use of DIS1.">" is legal since this operation is implicitly declared in DIS for AR, and so, is declared in DIS1. The use of and is illegal because no such operator is declared in DIS. The fact that DIS1.AR is an array of BOOLEAN is not relevant, since the operators declared for AR are determined solely by the template. Finally, note that the illegal use of and must be detected whether or not OUTER is ever instantiated.

S9. A careful analysis is also needed when a formal type has discriminants:

```

generic
  type T (A : INTEGER) is private;
package P is
  subtype S is T;
  TX : T(5);
  type NT is new T;
  NTX : NT(6);
  TXA : INTEGER := TX.A;  -- discriminant: Z1 = 5
  NTXA : INTEGER := NTX.A; -- discriminant: Z2 = 6
end P;

type REC (L : INTEGER) is
  record
    A : INTEGER := 7;
  end record;

package P1 is new P (REC);

X1 : P1.S(8);
X2 : P1.NT(9);

Y1 : INTEGER := X1.A;  -- Y1 = 7; component A
Y2 : INTEGER := X.L;   -- Y2 = 8; discriminant L

Y3 : INTEGER := X2.A;  -- Y3 = 9; discriminant A
Y4 : INTEGER := X2.L;  -- illegal

Y5 : INTEGER := P1.TX.A; -- Y5 = 7; component A
Y6 : INTEGER := P1.NTX.A; -- Y6 = 6; discriminant A

```

```

Y7 : INTEGER := P1.TX.L;      -- Y7 = 5; discriminant L
Y8 : INTEGER := P1.NTX.L;     -- illegal

```

To understand why these answers are correct, you must consider just what operations are implicitly declared for P1.S and P1.NT. Since P1.S is another name for the formal parameter, P1.S denotes REC. Therefore, X1 has type REC. Since the basic operations for REC are declared after REC's declaration (i.e., not in template P), and since these include operations for selecting component A and discriminant L, X1.A yields the value of component A. X2, however, has type P1.NT. The basic operations for NT are implicitly declared in template P, and the only component selection operation implicitly declared in the template is the operation to select discriminant A of the formal parameter. The fact that P1.NT's parent type is REC does not change the basic operations declared for NT inside P and P1. After P is instantiated, the component selection operation for T's discriminant denotes the component selection operation for the actual parameter's discriminant (RM 12.3/15). Hence, X2.A selects the discriminant component of X2; X2.L is illegal since there is no such component selection operation declared in P (or P1).

Similarly, P1.TX.A uses the selection operation for TX's type. Since TX's type is T, and since T denotes REC inside P1, P1.TX.A selects the component value of TX. Inside template P, TX.A invokes the selection operation for T's discriminant. The corresponding operation is used in P1, and so, after P1 is elaborated, P1.TXA = 5. Hence, P1.TX.A does not equal P1.TXA. Finally, P1.NTX.A uses the selection operation declared for NTX's type, P1.NT. This operation is implicitly declared after P.NT's declaration; it selects the discriminant of objects having type NT. This operation in P1, therefore, denotes REC's discriminant selection operation. Hence, P1.NTX.A equals 6. For similar reasons, P1.TX.L gives TX's discriminant value, and P1.NTX.L is illegal.

S10. Only operations declared in the visible part of an instantiated generic package can be named from outside the package (RM 4.1.3/14,15). This includes implicitly declared operations:

```

generic
  type T is range <>;
package P is
  X, Y, : T;
  type NT is new T;
  function "+" (L, R : NT) return NT;
end P;

package P1 is new P (INTEGER);
use P1;
... P1."+" ...      -- names redefined "+" for NT
... P1."-" ...      -- names implicitly declared "-" for NT
... X + Y ...       -- names STANDARD "+" for P1.T, i.e., INTEGER
... P1."+" (X, Y) ... -- illegal; no "+" for T declared in P's
                      -- visible part

```

Since T is an integer type, all the integer operations are declared implicitly for NT, and the nonbasic implicitly declared operations can be named using expanded names. Note that this notation is allowed inside generic units themselves:

```

generic
package Q is
  package P2 is new P (INTEGER);
  X : BOOLEAN := P2."<" (3, 4);

```

S11. For a formal fixed point type, multiplying operators are not declared immediately after the type declaration, but instead are declared in STANDARD (RM 12.1.2/13). Hence:

```
generic
  type FP is delta <>;
package P is
  X : FP := 3.0;
  Y : FP := FP (X*X);           -- legal
  Z : FP := FP (P."*" (X, X)); -- illegal
end P;
```

P."*" is illegal because there is no implicit declaration of "*" (or "/") after FP's declaration. P."+" (X, X) would, however, be legal.

S12. A generic formal type is not a static type (RM 4.9/11). The use of a generic formal type as an index subtype in an array type declaration implies that certain forms of aggregates are illegal:

```
generic
  type T is range <>;
package P is
  type ARR is array (T range <>) of INTEGER;
  W : ARR (1..4) := (1, others => 0);           -- illegal aggregate
  X : ARR (1..4) := (1 => 0, 2..4 => 1);         -- illegal aggregate
  Y : ARR (1..4) := (1..4 => 0);                 -- legal aggregate
end P;
```

The first aggregate is illegal because ARR's index subtype is T, which is a generic formal type, and so is nonstatic. An *others* choice must appear as the only choice of a single component association if the corresponding index subtype is nonstatic (RM 4.3.2/3). The second aggregate is illegal because an expression having a formal generic type is considered nonstatic. The choices, 1, 2..4, and 1..4 all have type T, and so are all nonstatic. If a choice is nonstatic, it must be the only choice of a single component association (RM 4.3.2/3). Since there are two component associations in the second aggregate, the aggregate is illegal. The last aggregate is legal because a single nonstatic choice is always legal. (See Language Maintenance Committee Commentary AI-00190.)

S13. Although a formal generic type is not static, the actual type may be static, and this can affect the legality of aggregates. Continuing with the previous example, consider the following:

```
subtype ST is INTEGER range 1..4;           -- ST is static
package ST_P is new P (ST);
Z : ST_P.ARR(1..4) := (1 => 0, 2..4 => 1);    -- legal
...
ST_P.X := (1, others => 0);                  -- legal
```

After the instantiation, the index subtype of ST_P.ARR is ST, which is static. Since ST_P.ARR has a static index subtype, the indicated aggregates are legal.

Changes from July 1982

S14. Each generic formal type is distinct from all other types.

S15. If a formal type has discriminants, no use of the unconstrained type mark is allowed where a constraint would normally be required.

S16. For a formal fixed point type, the multiplication and division operations are declared in STANDARD rather than after the formal type declaration.

Changes from July 1980

- S17. Operations of a formal type are implicitly declared immediately after the type declaration.
- S18. The operations declared for each class of type are now specified explicitly.
- S19. The attributes 'IMAGE and 'VALUE are implicitly declared for all scalar formal types.
- S20. Discriminant parts for formal types are no longer allowed to have default expressions.

Legality Rules

- L1. A range, accuracy, index, or discriminant constraint is forbidden when declaring the component type of a formal array type or when declaring a formal access type (RM 12.1.2/2).
- L2. Every discrete range specifying the bounds of a formal constrained array type must have the form of a type mark (RM 12.1.2/2).
- L3. A discriminant part for a formal type must not have a default expression (RM 12.1.2/3).

Test Objectives and Design Guidelines

- T1. Check that a generic formal type parameter with a generic scalar, array, or access type definition cannot have a discriminant part.
 Check that a range or accuracy constraint is forbidden when declaring a generic formal numeric type parameter.
 Check that default expressions are not permitted in a discriminant part of a formal private type.
 Check that a range, accuracy, index, or discriminant constraint is forbidden when specifying the component type of a formal array type or the designated type of an access type.
 Check that none of the following forms of discrete range are permitted when declaring the bounds of a formal constrained array type: L..R, T range L..R, and T'RANGE.
- T2. Check that a generic formal type parameter cannot have a record, derived, or incomplete type definition, and cannot be declared as an exception or a subtype.
- T3. Check that a generic formal type cannot be used recursively as a component type in an array type definition or as the type in an access type definition.
- T4. Check that formal private and limited private types may have discriminants, and that the type of the discriminant may be given by a previous generic formal parameter (of a discrete type).
- T5. Check that each formal type declaration declares a distinct type.
- T6. Check that an unconstrained formal type with discriminants is not allowed as the parent type in a full declaration of a private type (see IG 7.4.1/T4).
 Check that a constrained formal type with discriminants is allowed as such a parent type (see IG 7.4.1/T5).
- T7. Check that an unconstrained formal type with discriminants is not allowed in a variable declaration, a component declaration (of an array or record), or an allocator.
Implementation Guideline: Include a case where the component type is used as a component type of a formal array type.
 Check that an unconstrained formal type with discriminants is allowed as the type of a

subprogram or an entry formal parameter, and as the type of a generic formal object parameter, as a generic actual parameter, and in a membership test, in a subtype declaration, in an access type definition, and in a derived type definition.

Check that a formal type can be used to declare a component of an array or record, and that values can be correctly assigned to such components.

- T8. Check that the discriminants of a formal parameter must all have different identifiers.
- T20. For a formal discrete type, check that the following basic operations are implicitly declared and are therefore available within the generic unit: assignment, membership tests, qualification, explicit conversion, 'BASE, 'FIRST, 'LAST, 'SIZE, 'ADDRESS, 'WIDTH, 'POS, 'VAL, 'SUCC, 'PRED, 'IMAGE, 'VALUE.

Implementation Guideline: IG 4.9/T9 checks that the attributes of a formal discrete type are nonstatic.

Check that only the following nonbasic predefined operations are implicitly declared: relational operators.

Implementation Guideline: To check the availability of basic operations, each should be used within the generic unit, and the results checked for correctness when the unit is instantiated.

Implementation Guideline: To check on the existence of the nonbasic operations, each should be used within the generic unit and the results checked. In addition, a separate test should check that each can be named with an expanded name inside the unit. Finally, a type should be derived from a formal type declared in the visible part of a generic package, and selectively named from outside an instantiation of the package, even when the instantiation occurs inside another generic unit.

Implementation Guideline: To check that only these operations are available, try some operations that would be legal for the instantiated types but that are illegal for the formal type.

Implementation Guideline: Make the above checks when the unit is instantiated with an integer, a character, and an enumeration type that is not a character type.

- T21. For a formal integer type, check that the following basic operations are implicitly declared and are therefore available within the generic unit: all the basic operations for a discrete type (see IG 12.1.2/T20), explicit conversion to and from other numeric types, and integer literals (i.e., implicit conversion from *universal_integer* values).

IG 4.9/T9 checks that the attribute of a formal integer type are nonstatic.

Check that only the following nonbasic predefined operations are implicitly declared: relational operators and arithmetic operators (+, -, *, /, **, and abs).

Implementation Guideline: Check that the second operand of ** cannot be a formal integer type.

- T22. For a formal floating point type, check that the following basic operations are implicitly declared and are therefore available within the generic unit: assignment, membership tests, qualification, explicit conversion to and from other numeric types, real literals (implicit conversion from *universal_real* to the formal type), 'BASE, 'FIRST, 'LAST, 'SIZE, 'ADDRESS, 'DIGITS, 'MANTISSA, 'EPSILON, 'EMAX, 'SMALL, 'LARGE, 'SAFE_EMAX, 'SAFE_SMALL, 'SAFE_LARGE, 'MACHINE_RADIX, 'MACHINE_MANTISSA, 'MACHINE_EMAX, 'MACHINE_EMIN, 'MACHINE_ROUNDS, 'MACHINE_OVERFLOW.

Check that only the following nonbasic predefined operations are implicitly declared: relational operators and arithmetic operators (+, -, *, /, **, and abs).

- T23. For a formal fixed point type, check that the following basic operations are implicitly declared and are therefore available within the generic unit: assignment, membership tests, qualification, explicit conversion to and from other numeric types, and real literals (i.e., implicit conversion from *universal_real* to the formal type), 'BASE, 'FIRST, 'LAST, 'SIZE, 'ADDRESS, 'DELTA, 'MANTISSA, 'SMALL, 'LARGE, 'FORE, 'AFT, 'SAFE_SMALL, 'SAFE_LARGE, 'MACHINE_ROUNDS, 'MACHINE_OVERFLOW.

Check that only the following nonbasic predefined operations are implicitly declared: the

relational operators, * with one INTEGER operand, / with the second operand of type INTEGER, +, -, and abs.

Check that the operations for multiplying or dividing two fixed point values are not implicitly declared in the generic unit, i.e., check that these operators cannot be named by selection using a prefix that denotes the generic unit.

- T24. For array types with a nonlimited component type (of a formal and nonformal generic type), check that the following operations are implicitly declared and are therefore available within the generic unit: assignment, the operation associated with aggregate notation, membership tests, the notation associated with indexed components, qualification, explicit conversion, 'BASE, 'SIZE, 'ADDRESS, 'FIRST, 'FIRST(N), 'LAST, 'LAST(N), 'RANGE, 'RANGE(N), 'LENGTH, 'LENGTH(N).

Check that except for assignment and aggregate notation, the above operations are available even if the component type is limited.

Check that the nonbasic operations for equality and inequality are implicitly declared only if the component type is not limited.

For one-dimensional arrays, check that the following:

- basic operations are implicitly declared if the component type is:
 - limited or nonlimited: slicing;
 - a character type: string literals.
- nonbasic operations are implicitly declared if the component type is:
 - nonlimited: catenation, =, /=;
 - discrete (including a formal integer type and a formal discrete type): predefined relational operators;
 - boolean: not and the logical operators.

Check that no new operations are declared when a unit is instantiated.

- T25. For a formal access type, check that the following basic operations are implicitly declared and are therefore available within the generic unit: assignment, allocators for the access type (when the designated type is either a formal type or not and either has discriminants or does not), membership tests, qualification, explicit conversion, the literal null, selection with selector all, 'BASE, 'SIZE, 'ADDRESS, 'STORAGE_SIZE.

Check that only the following nonbasic predefined operations are declared: =, /=.

Check that the following basic operations are implicitly declared if the designated type is:

- a record type or a type with discriminants: selection of components of the designated type;
Implementation Guideline: Include a case where the designated type is a formal type with discriminants. Check that discriminant names of the designated types are used where the designated type is itself a formal type with discriminants.
- an array type: indexed component selection;
- a one-dimensional array type: slices;
- a constrained array type: 'FIRST, 'FIRST(N), 'LAST, 'LAST(N), 'RANGE, 'RANGE(N), 'LENGTH, 'LENGTH(N).

- a task type: selection of an entry or entry family.

T26. For a formal nonlimited private type, check that the following basic operations are implicitly declared: assignment, membership tests, qualification, explicit conversion, 'BASE, 'SIZE, 'CONSTRAINED, and 'ADDRESS.

For a formal limited private type, check that except for assignment, all the above basic operations are implicitly declared.

For a formal limited or nonlimited private type with discriminants, check the following basic operations are declared: selection of a discriminant for each discriminant and 'CONSTRAINED.

Implementation Guideline: Check that the discriminant selection operations use the names of the formal type, not the names of the actual type (see also IG 12.3.2/T40).

Check that no nonbasic operations are declared for a formal limited private formal type.

Check that equality and inequality are implicitly declared for a formal nonlimited private type.

T27. Check that when deriving from a formal type, all and only the predefined operations associated with the class of the formal type are declared for the derived type (cf. IG 12.3.2/T20-T40).

T30. Check that a formal type can only be used after its declaration.

Implementation Guideline: Include a case where there is an external type declaration with the same identifier.

12.1.3 Generic Formal Subprograms

Semantic Ramifications

S1. The visibility rules for default names in formal subprogram declarations are the same as for any other name in a generic unit, namely, a declaration of the name must be visible at the place where the name is used in the generic declaration. The visibility of names at the place of an instantiation is irrelevant. Consequently, if a generic formal subprogram declares a parameter or identifier using a generic type previously defined in the same generic formal part, then no identifier, selected component, or indexed component form of name is legal as a default name, since no subprogram outside the generic formal part can match the formal subprogram specification:

```
generic
  type T is ...;
  with procedure P (X : T) is D;    -- illegal
package Q is ...
```

In the above example, T can be any generic type definition; there cannot be any D declared outside the generic specification with a formal parameter of type Q.T, and so the default name is illegal.

S2. However, if there is a preceding *formal* declaration of D:

```
generic
  type T is private;
  with procedure D (X : T);
  with procedure P (X : T) is D;    -- now legal
package ...
```

the D in P's declaration is identified with the preceding formal subprogram.

Implicitly declared operators can be named this way as well:

```
generic
  type T is private;
  with function EQU (L, R : T) return BOOLEAN is "=";
package ...
```

The default for EQU is the predefined "=" implicitly declared for T. Note that the restrictions of RM 6.7/4 do not apply here since EQU is not an equality operator.

S3. The situation is similar when generic units are nested inside generic units:

```
procedure P is
  generic
    type T is private;
  procedure Q:

  procedure Q is
    procedure TARGET (X : T) is ... end;          -- 1
  begin
    declare
      procedure TARGET (X : FLOAT);                -- 2

      generic
        with procedure FOO (X : T) is TARGET;      -- 3
      procedure R:
        ...
      begin ... end;
    end Q;

  procedure N_Q is new Q (FLOAT);
```

The default subprogram at 3 is identified with the subprogram declared at 1, since this is the only visible TARGET subprogram with a parameter of type T. The fact that the later instantiation identifies T with FLOAT does not change the default subprogram identification at 3 -- the identification of names is determined by the visibility rules, and so occurs prior to the elaboration of a template. (Note that the template for R is not elaborated until Q is called.)

S4. It is always legal to use a box as a default subprogram parameter. When a box is used, the default actual parameter is determined at the point of instantiation (see IG 12.3.6/S). For example,

```
generic
  type T is range <>;
  type U is range <>;
  with function PLUS (L, R : T) return T is "+";  -- predefined "+"
  with function "+" (L, R : U) return U is <>;
  with function "+" (L, R : T) return T is "+";  -- illegal
```

The default for PLUS is the "+" operator implicitly declared for formal parameter T, and this is the predefined "+" operation. The explicit declaration of "+" for type U hides the implicit declaration of predefined "+" for U. When the generic unit is instantiated, the formal declaration of "+" will denote whatever "+" operation is visible for U's actual parameter. The last declaration of "+" is illegal because the declaration of the "+" function hides all other declarations of "+" having the same parameter and result type profile (RM 8.3/15), and the declaration itself is not yet visible (RM 8.3/15); consequently, there is no visible "+" that has the required parameter and result type profile.

S5. An attribute is one form of name permitted as a default name when the formal subprogram contains a parameter of a generic type. However, only those attributes considered to be functions are permitted in this position, namely, 'IMAGE, 'VALUE, 'PRED, and 'SUCC (RM A). These attributes are only defined for discrete types. Hence, a generic declaration is illegal if these attributes are applied to a formal parameter that is not a discrete type, e.g.:

```
generic
  type T is private;
  with function SUCC (X : T) return T is T'SUCC;    -- illegal
  ...
```

S6. An entry can match the default name of a generic formal procedure:

```
task T is
  entry E (X : INTEGER);
end T;

generic
  with procedure P (Y : INTEGER) is T.E;    -- matches entry of T
  ...
```

Within the generic unit, P is considered a procedure and cannot be used in contexts that are only legal for entry calls, i.e., P cannot be used in a timed or conditional entry call.

S7. It is only possible for a default subprogram to contain an expression or to require the evaluation of an access value if the default name denotes an entry or a member of an entry family:

```
generic
  with procedure P is AR(I).E;    -- entry of array of tasks
  with procedure Q is T.E(I);    -- member of entry family
  with procedure R is ACC_TSK.E  -- access to entry of task
package P is ... end;
```

In all cases, the evaluation of the name (see RM 4.1/9) is performed only when the default parameter is needed in an instantiation (RM 12.3.6/2).

S8. Calls of a formal subprogram (i.e., calls occurring within the template) use the defaults specified in the formal subprogram's declaration, independent of any defaults that may be specified for the actual parameter. RM 6.4.2/1-2 implies that calls naming subprogram P use the default expressions specified in P's declaration even when P denotes a subprogram originally declared under a different name (and possibly with different defaults):

"If a parameter specification includes a default expression for a parameter of mode in, then corresponding subprogram calls need not include a parameter association for the parameter. ... For any omitted parameter association, the default expression[s] ... value is used as an implicit actual parameter."

Since a parameter_specification occurs only in a subprogram_specification, the "corresponding subprogram" is the subprogram whose declaration is named in the call as opposed to the subprogram denoted by the name in the call. Hence, the default expressions of the named subprogram's declaration are used as opposed to the default expressions (if any) of the denoted subprogram. For most calls, the denoted and the named subprogram are the same. But this is not the case when the named subprogram is declared by a renaming declaration or by a formal generic parameter declaration.

S9. An implementation must keep in mind that although the default expressions of a formal subprogram, P, are used when the name P is used in a call, the subtypes of P's parameters are

determined by the subprogram P denotes, not by the subtypes specified in P's declaration. (RM 12.3/f says that after instantiation, a formal subprogram parameter denotes its actual parameter. Since the subtype constraint checks required when calling a subprogram (see RM 6.4.1/5-10) are determined by the subtypes of the denoted subprogram's parameters, the required subtype checks are determined by the subprogram denoted by the parameter, not by the formal subprogram's parameter subtypes.) Hence:

```

subtype R_TEN is INTEGER range 1..10;
subtype R_FIFTY is INTEGER range 1..50;

procedure R (X : R_TEN);

generic
  type T is range <>;
  with procedure P (X : T := T'LAST);
procedure GQ;

procedure GQ is
begin
  P;          -- equivalent to P (T'LAST)
end;
```

When GQ is instantiated, the call to P can raise CONSTRAINT_ERROR:

```

procedure G1 is new GQ (R_FIFTY, R);
...
G1;          -- CONSTRAINT_ERROR raised
```

Within G1, P denotes R, so the parameterless call to P inside G1 is equivalent to R (R_FIFTY'LAST). Clearly R_FIFTY'LAST lies outside the range of R_TEN, so CONSTRAINT_ERROR must be raised. In short, as for renaming declarations, the parameter subtype of a formal subprogram does not determine what subtype checks must be made when the formal (or renamed) subprogram is called.

S10. When a default name is given for a formal subprogram, the parameter and result type profile of the formal subprogram is used to resolve any overloads of the given name (RM 12.3.6/1, RM 8.7/7, RM 8.7/19, and IG 8.7.b/S). After this resolution, the modes of the named subprogram are checked against the modes of the formal subprogram. If the modes do not match, the default name is illegal. The modes of the parameters are not used to help decide what subprogram is denoted by the default name, nor are the names of the formal parameters or the existence of any default expressions (RM 12.3.6/1):

```

package P1 is
  procedure P (X : INTEGER);
end P1;

package P2 is
  procedure P (Y : in out INTEGER);
end P2;

use P1, P2;

generic
  with procedure Q (Z : in INTEGER) is P; -- illegal
package R is
```

The default name *P* is illegal. Both *P1.P* and *P2.P* are visible (because of the use clauses; see RM 8.4/6) and they have the same parameter and result profile. The difference in the parameter modes cannot be used to decide which *P* is meant. If the default name were *P1.P*, the name would be legal; the modes are the same even though the mode is indicated explicitly in the declaration of *Q* (i.e., the conformance rules of RM 6.3.1 do not apply). Finally, the default name *P2.P* would be illegal because the parameter modes of *P2.P* and *Q* are not the same.

S11. Use clauses provide one way to make subprograms with equivalent profiles simultaneously visible (see IG 8.4/S). Another way is by instantiating a generic unit (see IG 12.3/S).

Changes from July 1982

S12. An enumeration literal is allowed as an actual parameter associated with a formal subprogram declaration.

Changes from July 1980

S13. A default subprogram specified by name is associated with a name visible at the point of the given unit's declaration, not at the point where the unit is instantiated.

Legality Rules

- L1. If a name is specified as the default name for a generic formal subprogram, there must be exactly one matching subprogram visible at the point of the generic declaration. (A visible subprogram matches the default name if the visible name and the number of parameters are the same, the parameters are in the same order, and corresponding parameters have the same base type.) (RM 12.3.6/1.)
- L2. The mode of each parameter of a subprogram denoted by a default name must be the same as the mode of corresponding parameter of the formal subprogram (12.3.6/1).
- L3. A default name must not be the same as the name of the formal subprogram (RM 8.3/5, 15).
- L4. A generic formal subprogram must not use any of the following strings as its designator: "in", "not in", "and then", "or else" (RM 6.7/1).
- L5. A generic formal function must only have parameters of mode *In* (RM 6.5/1).
- L6. The identifiers of a generic formal subprogram's parameters must be distinct from each other (RM 8.1/2 and RM 8.3/17).
- L7. The designator of a formal subprogram must not appear as a name within the subprogram's specification (RM 8.3/16).
- L8. Default values may only be specified for generic formal subprogram parameters of mode *in* (RM 6.1/4) and only for subprograms that do not declare operators (see RM 6.7/2).
- L9. The base type of the default expression must be the same as the base type of the formal parameter (RM 6.1/4).
- L10. A default expression for a formal parameter of a generic formal subprogram must not use a name denoting any formal parameter declared previously in the same formal part (RM 6.1/5), nor may the name of the current parameter be used (RM 8.3/5 and RM 6.1/5).
- L11. Operator symbols declared as generic formal functions having two parameters must correspond to operators that require two operands (RM 6.7/2).
- L12. Operator symbols declared as generic formal functions having a single parameter must correspond to operators that require one operand (RM 6.7/2).

- L13. If "=" is the designator of a generic formal function, its formal parameters must have the same limited type, and its result type must be predefined BOOLEAN (RM 6.7/4).
- L14. The generic formal part and the declarative part of a generic subprogram cannot contain equivalent explicit subprogram declarations (RM 8.3/17 and RM 8.1/2). (Two subprogram declarations are equivalent if (RM 6.6):

- they both declare procedures or both declare functions; and
- the number, order, and base types of the formal parameters are the same; and
- for functions, the result base types are the same.)

- L15. For generic packages, subprograms declared in a generic formal part, a package specification, and a package body must not be equivalent (RM 8.3/17 and 8.1/2).

Test Objectives and Design Guidelines

- T1. Check that default names of subprogram parameters are identified with subprograms visible at the point of the generic declaration, not at the point of the generic instantiation.

Implementation Guideline: Check defaults specified with identifiers (including enumeration literals), operator symbols, and character literals.

Implementation Guideline: For one case, the default subprogram should be visible via a use clause at the point of the declaration, but not visible at the point of instantiation.

Implementation Guideline: At least one default should match the name of a preceding generic formal parameter, whether it is explicitly or implicitly declared.

Check that no match to a default subprogram name is found if the only difference is in:

- the number of parameters;
- the ordering of the parameter base types;
- the mode of a parameter;
- the base type of a parameter;
- more than one matching subprogram is visible.

Implementation Guideline: Include a case like that discussed in S1.

Check that if an overloaded identifier or operator is used as a default subprogram, the overloading is resolved without considering the names of parameters, whether they have default values, and whether the mode in is written explicitly in both declarations.

Check that a default name cannot be the same as the formal parameter name.

Implementation Guideline: Use both simple names and operator symbols.

- T2. Check that generic default subprograms may be the following attributes, and that the appropriate functions are used: 'IMAGE, 'VALUE, 'PRED, and 'SUCC.

Implementation Guideline: Check using types declared in the same and in an enclosing generic formal part.

Check that default generic subprograms cannot be the following attributes:

' ADDRESS	' FORE	' MANTISSA
' AFT	' LARGE	' POS
' BASE	' LAST	' POSITION
' CALLABLE	' LAST (N)	' RANGE
' CONSTRAINED	' LAST_BIT	' SAFE_EMAX
' COUNT	' LENGTH	' SAFE_LARGE
' DELTA	' LENGTH (N)	' SAFE_SMALL

' DIGITS	' MACHINE_EMAX	' SIZE
' EMAX	' MACHINE_EMIN	' SMALL
' EPSILON	' MACHINE_MANTISSA	' STORAGE_SIZE
' FIRST	' MACHINE_OVERFLOW	' TERMINATED
' FIRST(N)	' MACHINE_RADIX	' WIDTH
' FIRST_BIT	' MACHINE_ROUND	' VAL

Implementation Guideline: In each case, use a generic formal subprogram whose parameter and return type are defined appropriately.

T3. Check that the usual restrictions for subprogram declarations are enforced for formal subprogram declarations, namely:

- a. "/"=, "in", "not in", "and then", "or else", and "!=" are forbidden as designators for generic formal functions;
- b. generic formal functions cannot have parameters of mode in out or out;
- c. parameter names of generic formal subprograms cannot be identical;
- d. default parameter values are forbidden for generic formal subprogram parameters of mode in out or out, and for formal subprograms whose designator is an operator symbol;
- e. the base type of a generic formal subprogram parameter and the base type of its default value cannot be different;
- f. the designator of the subprogram must not appear as a name within the subprogram specification;
- g. operator symbols declared as generic formal functions must have the correct number of parameters;
- h. the case of literals used as operator symbols is not significant;
- i. if "=" is a generic formal function, its formal parameters must have the same limited type and its result type must be BOOLEAN;
- j. formal subprogram declarations having the same designator and equivalent parameter and result profiles are forbidden within a generic formal part and between a generic formal part and either the specification and body of a package or the declarative part of a generic subprogram; in particular, check that parameter mode, default values, and presence or absence of constraints is ignored (see IG 12.1/T9).

T4. Check that generic formal subprograms may have parameters of a generic formal type.

Implementation Guideline: Include all modes. IG 7.4.4/T1 covers the case when the generic formal type is limited private and the parameter mode is out.

T5. Check that default subprograms may be instances of generics.

T6. Check that a formal subprogram cannot be used in a conditional or timed entry call, even if the generic unit is never instantiated.

T7. Check that a string literal may be used both as a default subprogram value (e.g., "=") and as a default initial value for an array parameter, and that the proper default values are used in an instantiation.

Implementation Guideline: The array parameter must be a one-dimensional array of a character type.

Check that an enumeration literal (both an identifier and a character literal) may be used as a default subprogram name and as a default initial value for an object parameter.

- T8. Check that formal subprogram parameters may overload each other and other visible subprograms and enumeration literals within and outside of the generic unit.

Implementation Guideline: Check using subprograms with minimal differences, i.e., use declarations that differ in only one of the following aspects:

- one is a function; the other is a procedure.
- one subprogram has one less parameter than the other (the omitted parameter may or may not have a default value).
- the base type of one parameter is different.
- the parameters are ordered differently.
- the result types of two functions are different.

- T10. Check that a default subprogram may be an entry or a member of an entry family, and that the appropriate entry is used.

Implementation Guideline: For some cases, the entry should be an entry of a task that is: designated by an access value, a component of a record, and a component of an array. The particular task or entry should be determined at the time the default is used.

Check that a default subprogram cannot be an entry family.

- T11. Check that the default expressions of the parameters of a formal subprogram are used instead of the defaults (if any) of the actual subprogram parameter.

Implementation Guideline: The default expressions should be evaluated when used in a subprogram call (rather than when the generic unit is declared or instantiated).

Check that if parameters of default and formal subprograms have the same base type but not the same subtype, the parameter subtypes of the subprogram denoted by the default are used instead of the subtypes specified in the formal subprogram declaration (see also IG 12.3.6/T5, which checks for actual parameters rather than default parameters).

Implementation Guideline: Use subtypes of scalar, record, array, and access types — both constrained and unconstrained.

12.2 Generic Bodies

Semantic Ramifications

S1. A generic formal part cannot be provided for a generic body. Thus a generic subprogram must have a separate declaration and body; the body can never serve to declare a generic unit (although bodies can serve as declarations for nongeneric subprograms; see RM 6.3 and RM 10.1/3).

S2. Since a generic declaration and its body form a single declarative region (RM 8.1/2), the scope of all declarations occurring immediately within a generic specification extends over the associated body (RM 8.2/2). The visibility rules (RM 8.3) therefore imply that names within a template body are statically bound, first using the context of the generic specification, and then using the context of the body (see IG 12.1/S).

Changes from July 1982

S3. There are no significant changes.

Changes from July 1980

S4. There are no significant changes.

Legality Rules

- L1. A subprogram body must be provided for each generic subprogram declaration (RM 12.2/2).

L2. A package body must be provided if any of the following are given as a declarative item of a generic package specification (RM 7.1/4):

- a nongeneric subprogram declaration (RM 6.3/3), unless the subprogram is named in an INTERFACE pragma that is accepted by the implementation (see RM 13.9/3).
- a generic subprogram declaration (RM 12.2/2).
- a task declaration (RM 9.1/1).
- a (nested) package declaration or generic package specification that requires a body (RM 7.1/4).
- an incomplete type declaration in the private part of a package without a corresponding full type declaration in the same private part (RM 3.8.1/3).

L3. The body of a generic unit must not precede its declaration (RM 3.9/9).

Test Objectives and Design Guidelines

T1. Check that a generic subprogram or package body cannot be preceded by a generic formal part, even when the generic formal part is copied from the corresponding generic declaration.

Implementation Guideline: In some cases, there should be no preceding generic subprogram declaration.

T2. Check that the statements in a generic package body are not executed when the generic body is elaborated.

Implementation Guideline: The elaboration of a nongeneric package body can change the values of global variables and/or raise exceptions. Neither of these should occur when a generic package body is elaborated; only when it is instantiated.

T3. Check that names in a generic body are bound in the context of the generic declaration (not the instantiation) (see IG 12.1/T10).

T4. Check that the body of a generic unit must appear after its declaration.

12.3 Generic Instantiation

Semantic Ramifications

S1. Detailed matching rules for the various kinds of generic parameters are discussed in later subsections, as in the RM. This section covers the remaining aspects of generic instantiations.

S2. Although generic units are like macros in many ways, there are important differences; an implementation cannot treat a generic unit as a macro because:

- the actual parameters of an instantiation are evaluated (once) in the context of the instantiation, not in the context of their use within the instantiated unit;
- an instantiation provides (and, for packages, executes) the body of the instantiated unit at the point of the instantiation, even though it may be syntactically illegal to actually provide a body at that point (e.g., within a package specification);
- name binding and the effect of deriving a type (see IG 12.1.3/S) are all defined within the context of the generic declaration, not in the context of the instantiation;
- the instantiated unit may contain homographic declarations that would be illegal

if the text of the instantiated unit were written at the place of the instantiation (see later example).

S3. In general, the legality of a generic instantiation only depends on:

- a. whether the actual parameters of the instantiation satisfy restrictions associated with the formal parameters;
- b. whether the name of the instantiated unit is allowed in the declarative region containing the instantiation (see RM 8.3/17).
- c. the use within the template of a formal private or limited private type when the actual type is:

- an unconstrained type having discriminants, or
- an unconstrained array type (see IG 12.3.2/S).

- d. whether the instantiation introduces a circularity.

S4. The legality of an instantiation depends on the contents of a template only in cases (c) and (d). If an implementation notes how formal private parameters are used in a generic unit and what instantiations such a unit performs, this information, combined with the properties of the formal parameters and the name of the instantiated unit, suffice to determine the legality of the instantiation. One difficulty is that an instantiation may textually occur before the body of the instantiated unit, and can even occur in a separately compiled unit. The problems this causes are discussed later in conjunction with (c) and (d).

S5. The identifier declared by a package instantiation is not visible within the instantiation (RM 8.3/5), i.e., any outer declarations with that identifier are hidden and can only be named selectively:

```
package NP is new P (NP.T);           -- illegal
package NQ is new Q ((NQ => 0));      -- legal
```

The declaration of NQ is legal on the assumption that Q's formal parameter is a record type with component name NQ.

S6. Generic subprogram instantiations obey a stricter rule: the designator of the instantiated subprogram cannot be used within the instantiation to denote any declaration, either directly or by selection (RM 8.3/16):

```
function F return FLOAT is ... end F;

generic
  type T is private;
  X : FLOAT;
function G return T;

function G return T is ... end G;

with F, G;
package P is
  X : FLOAT := F;                      -- invokes library unit
  type REC is
    record
      F : FLOAT := 3.0;
    end record;
```



```

R : REC;

function F is
  new G (INTEGER, F);           -- illegal; STANDARD.F hidden
function F is                   -- illegal; F not visible
  new G (FLOAT, STANDARD.F);    -- by selection
function F is
  new G (INTEGER, P.F);         -- illegal; P.F not visible
function F is
  new G (INTEGER, R.F);         -- illegal; no F is visible
end P;

```

All the instantiations of F are illegal because within each instantiation, the identifier F appears. RM 8.3/16 says, "Within a generic instantiation that declares a subprogram, ... every declaration with the same designator as the subprogram is hidden; ... where hidden in this manner, a declaration is visible neither by selection nor directly." In each of the cases shown above, an attempt is made to reference a declaration of F within a generic instantiation of subprogram F. All such attempts are illegal.

When an instantiated function is denoted by an operator symbol, this restriction extends to the use of the operator symbol in a function call and to the use of the operator in an expression:

```

function "+" is new G(
  "+" (3.2), -- illegal use of "+"
  3 + 2,     -- illegal use of +
  "+");      -- illegal if formal parameter
              -- is a function

```

The last usage is not illegal if "+" denotes a string literal (e.g., if the formal parameter has the type STRING (1..1)), since in such a case, "+" does not denote any declaration.

S7. When an instantiated function is declared with an operator symbol, the rules for declaring operators apply (RM 6.7/2), i.e., default parameters are not allowed, the operator must have the correct number of parameters, and special rules apply to the declaration of "=".

```

generic
  type T is private;
  DEFAULT : T;
function BIN_OP (L, R : T := DEFAULT) return T;

function "*" is new BIN_OP (INTEGER, 3); -- illegal; default values
function "abs" is new BIN_OP (INTEGER, 3); -- illegal; 2 parameters

```

The first instantiation is illegal because it provides default parameter values for the operands of "*". The second is illegal because it attempts to provide two parameters for a unary operator.

S8. Since default values of generic formal parameters can depend on the values of previous formal parameters, default expressions must be evaluated in the order of their occurrence in the generic unit's declaration:

```

generic
  X : INTEGER := 3;
  Y : INTEGER := X - 1;
  Z : INTEGER := X + Y;
package P is ... end P;

```

```

package NP1 is new P (Y => 6);      -- X = 3; Z = 9
package NP2 is new P (5);          -- X = 5; Y = 4; Z = 9

```

S9. As a result of an instantiation, it is possible to obtain two subprograms with identical parameter and result type profiles:

```

generic
  type T1 is private;
  type T2 is private;
package P is
  procedure PROC (X : T1);
  procedure PROC (X : T2);
end P;

package PI is P (INTEGER, INTEGER);

```

The instantiated package contains two subprograms with identical profiles. This does not make the instantiation illegal, but it does mean that any calls to PI.PROC will be ambiguous, and therefore, illegal. Note that within the body of P, PROC can be called unambiguously with parameters of type T1 or T2, since the visibility rules bind such uses of PROC in the context of the generic declaration:

```

package body P is
  procedure PROC (X : T1) is ... end PROC;
  procedure PROC (X : T2) is ... end PROC;
  Z : T1;
begin
  PROC (Z);      -- where Z is of type T1
end P;

```

S10. Circular instantiations are forbidden:

```

generic procedure F;
generic procedure G;

with G;
procedure F is
  procedure NG is new G;
begin
  NG;
end F;

with F;
procedure G is
  procedure NF is new F;      -- illegal
begin
  NF;
end G;

```

The instantiation of F is illegal regardless of whether G is ever instantiated. An instantiation of F can legally occur prior to the compilation of G, and F and G can both be separately compiled as library units or subunits. For example, suppose unit P, below, is compiled after the compilation of unit F's body:

```

with F;
package P is
    procedure NF is new F;
end P;

```

Unit P is legal; only G, the unit that completes the circularity, is illegal. Note that an instantiation that creates a circularity can only appear inside a generic unit.

S11. The circularity rule does not take potential flow of control into account. If we replace the above body of G with:

```

procedure G is
begin
    if FALSE then
        declare
            procedure NF is new F;    -- still illegal
        begin
            null;
        end;
    end if;
end G;

```

there is still an illegal circularity.

S12. In general, a generic unit body can be compiled or recompiled after compiling several units that contain instantiations of the unit. Recompiling such bodies does not require recompilation of the units containing the instantiations. See IG 10.3/S for further discussion.

Changes from July 1982

S13. An actual parameter for a formal generic subprogram may denote an enumeration literal or entry.

S14. The evaluation order for default expressions is defined.

Changes from July 1980

S15. The mapping from formal parameters and implicit operations to actual parameters and operations is no longer defined in terms of a replacement rule.

S16. An operator symbol is allowed in a named parameter association.

Legality Rules

- L1. The name following *new* in a generic instantiation must be the name of a generic subprogram or package (in particular, it must not be the name of an enclosing generic unit) (RM 12.3/2).
- L2. A generic actual parameter must be supplied for each generic formal parameter that has no default given in the corresponding generic formal part (RM 12.3/3).
- L3. The number of generic actual parameters must not exceed the number of generic formal parameters in the corresponding generic formal part (RM 12.3/3).
- L4. A named generic actual parameter must not be given for a generic formal parameter that is already associated with an earlier positional or named generic actual parameter (RM 12.3/3 and RM 6.4/4).
- L5. Named associations are not allowed for overloaded formal parameters (RM 12.3/3).
- L6. Positional generic actual parameters must precede named generic actual parameters (RM 12.3/3 and RM 6.4/4).

- L7. The formal parameter name in a generic association must be identical to that of a generic formal parameter in the corresponding generic formal part (RM 12.3/3 and RM 6.4/3).
- L8. Circular instantiations are forbidden, i.e., if a generic unit, A, is said to statically instantiate unit B when A's text contains an instantiation of B or an instantiation of a unit that statically instantiates B, a generic unit is forbidden to statically instantiate itself (RM 12.3/18).
- L9. Any declaration having the designator given in an instantiation of a subprogram is not directly or selectively visible within the instantiation (RM 8.3/16).
- L10. Any declaration having the identifier given in an instantiation of a package is not directly visible in the instantiation (RM 8.3/5).
- L11. If an instantiation declares an operator:
 - the instantiated unit must not have a default parameter (RM 6.7/2);
 - if the operator is a binary operator, the instantiated function must have two parameters (RM 6.7/2);
 - if the operator is a unary operator, the instantiated function must have one parameter (RM 6.7/2);
 - if the operator is equality, the formal parameters of the instantiated function must both have the same limited type and the function must deliver a BOOLEAN result (RM 6.7/4);
 - the designator must not be "/=" (RM 6.7/4), "in", "not in", "and then", or "or else" (RM 6.7/1).

Test Objectives and Design Guidelines

- T1. Check that a reference to an instantiation of a generic subprogram or package cannot precede the instantiation itself.

Check that the identifier of a package declared by an instantiation is not directly visible within the instantiation.

Implementation Guideline: Check that the identifier cannot be used as the name of the generic unit being instantiated or in the actual parameter list as a type mark; as a primary in an expression; as a function name in a function call; as a prefix of an indexed component, slice, selected component, or attribute; or as a selector of an expanded name whose prefix denotes a unit immediately enclosing the instantiation.

Check that the identifier of a package is selectively visible within the instantiation.

Implementation Guideline: Check that the identifier can be used as the selector of an expanded name whose prefix denotes a visible package or a construct enclosing the instantiation (but not a construct immediately enclosing the instantiation); as a selector denoting a record component; as a choice in a record aggregate; as a formal parameter name in a function call; as a selector denoting an entry or entry family; or as a formal parameter name of the unit being instantiated.

Check that the designator of a subprogram declared by an instantiation is not visible directly or selectively within the instantiation.

Implementation Guideline: Use all the contexts specified above for a generic package instantiation.

- T2. Check that in a generic instantiation:

- a generic actual parameter cannot be omitted if the corresponding generic formal (object or subprogram) parameter has no default.
- the number of generic actual parameters cannot exceed the number of generic formal parameters in the corresponding generic formal part.
- named generic actual parameters cannot precede or be interleaved with positional generic actual parameters.

- a positional or named generic actual parameter and another named generic actual parameter cannot be specified for the same generic formal parameter.
- the formal parameter identifier in a named generic association cannot be different from all of the identifiers of the generic formal parameters in the corresponding generic formal part.

Implementation Guideline: When possible, try each of the above with generic instantiations consisting of (1) positional associations only, (2) named associations only, and (3) mixtures of positional and named associations.

- T3. Check that generic instantiations of the form $P(A, B)$ are forbidden, even when the omitted generic parameter has a default.

Check that generic instantiations of the form $P(A|B \Rightarrow C)$ are forbidden, even when generic formal parameters A and B are of the same kind (object, type, or subprogram) and match (same subtype, same kind of generic type definition, or same subprogram specification).

- T4. Check that regardless of the order of the named generic actual parameters in a generic instantiation, each is associated with the generic formal parameter that has the same formal parameter identifier or operator symbol.

Check that all actual parameters are evaluated prior to any default expression.

Check that defaults for formal parameters are evaluated in the order given by the generic declaration (see IG 12.3.1/T23 and IG 12.3.6/T2).

Check that default expressions are only evaluated if no actual parameter is present (see IG 12.3.1/T23 and IG 12.3.6/T1, /T2).

- T5. Check that in a generic instantiation:

- the actual parameter corresponding to a formal type or a formal subprogram cannot be an object.
 - the actual parameter corresponding to a formal object or a formal subprogram parameter cannot be a type mark.
 - the actual parameter corresponding to a formal type cannot be a subprogram or an entry name, nor can it be a subtype indication with a range constraint, an accuracy constraint, an index constraint, or a discriminant constraint.
- Implementation Guideline:* Try all classes of formal type.
- the actual parameter corresponding to a formal object cannot be a type name, a procedure name, or an entry name.

- T6. Check that the name following `new` in a generic instantiation cannot be the name of an instantiation or the name of a nongeneric package or subprogram.

- T7. Check that the names in a generic instantiation are statically identified (i.e., bound) at the textual point of the instantiation, and are bound before being "substituted" for the corresponding generic formal parameters in the specification and body templates.

Implementation Guideline: Use tests that distinguish between LISP/APL/macro dynamic binding and Algol/Pascal/Ada static binding, such as when a generic actual parameter identifier is identical to a local identifier of a template and is used in the local identifier's declarative region.

- T8. Check that a generic instantiation within a generic unit is performed correctly. (This objective is accomplished implicitly by other tests.)

- T9. Check that a generic unit may not require an instantiation of itself.

Implementation Guideline: Use directly and indirectly circular instantiations, both when all the bodies are declared in the same compilation unit, and when the bodies are declared in different compilation units (but within the same compilation); use both subunits and library units. Also check that a circularity is detected even if an instantiation or a generic declaration is not elaborated.

T10. Check that for instantiations that declare an operator (see IG 6.7/T1):

- no default parameters are allowed;
- binary operators must have two parameters;
- unary operators must have one parameter;
- the equality operator must have two parameters of the same limited type, and must have a predefined BOOLEAN result type.

Check that instantiations of the above operators are allowed and can be used appropriately (see IG 6.7/T2).

T11. Check that an instantiation with identical actual type parameters can produce subprogram declarations having the same type profile in the instantiated unit, and that calls within the instantiated unit are unambiguous.

Check that calls from outside the instantiated unit are unambiguous if formal parameter names are used or if only one of the equivalent programs appears in the visible part of an instantiated package, and otherwise are ambiguous.

T12. Check that instantiated generic subprograms can overload previously declared subprograms and enumeration literals.

T13. Check that a specification part cannot be provided in the instantiation of a generic subprogram.

T14. Check that " /= ", "in", "not in", "and then", and "or else" cannot be declared by a generic instantiation (see IG 6.7/T1).

T15. Check that when a generic package instantiation is elaborated, statements in its package body are executed and expressions requiring evaluation are evaluated (e.g., defaults for object declarations are evaluated).

T16. Check that an instantiated package has the properties required of a package (see tests for Chapter 7).

T17. Check that an instantiated subprogram has the properties determined by the generic unit's specification and body (see tests for Chapter 6).

T18. Check that named associations are not allowed for overloaded formal parameters.

T19. Check instantiations of units within generic units, e.g., to support iterators.

12.3.1 Matching Rules for Formal Objects

Semantic Ramifications

S1. Subcomponents depending on a discriminant of an unconstrained (containing) object are not allowed as generic actual in out parameters (RM 12.3.1/2 and 8.5/5). Such a containing object always has a record type with default discriminants. (The object cannot have a scalar or access type, since such types have no subcomponents. It can't have an array type or a type without default discriminants, since all such variables must be constrained. The only type classes left are record and private types that have default discriminants. Since discriminants

are the only visible subcomponents of a private type, and since discriminants cannot themselves be declared so they depend on another discriminant, the only kind of unconstrained object that can have subcomponents depending on a discriminant is a record object whose type has default discriminants.)

S2. The containing object itself can be (see RM 8.5/5):

- a variable declared in an object declaration;
- a component of a record variable;
- a component of an array variable;
- an in out formal parameter of a subprogram or entry; or
- a generic formal in out parameter (whether or not it is constrained).

In all but the last case, the object must have an unconstrained subtype with discriminants that have defaults. (The case of generic formal variables is discussed later.) According to RM 3.7.1/6-7, a component of such a variable depends on a discriminant if:

- the component is declared in a variant part;
- the component is an array with one or more bounds specified by a discriminant (of the enclosing record);
- the component has a record or private type and is declared with a discriminant constraint that names a discriminant of the enclosing record; or
- the component has an access type and is declared with an explicit (index or discriminant) constraint naming a discriminant of the enclosing record.

Any subcomponent of such a component is also considered to depend on a discriminant. However, since access types do not have (sub)components, a designated object (and its components, if any) does not depend on a discriminant even if the access variable does depend on a discriminant (see example below).

S3. The various possibilities are illustrated in the next example:

```

type REC (D : INTEGER := 0) is
  record
    A : INTEGER;
    case D is
      when INTEGER =>
        V : INTEGER;           -- decl in variant part
    end case;
  end record;

type AR_REC is array (1..10) of REC;
type R_REC is
  record
    E : REC;
  end record;

type A_STRING is access STRING;
type A_REC is access REC;
type A_AR_REC is access AR_REC;
type A_R_REC is access R_REC;

```

```

type DIS (L : POSITIVE := 1) is
  record
    S      : STRING (1..L);          -- dependence on discriminant
    R      : REC (L);                -- dependence on discriminant
    RC     : REC (3);
    AS     : A_STRING (1..L);        -- dependence on discriminant
    AR     : A_REC (L);              -- dependence on discriminant
    ARU    : A_REC;                  -- dependence on discriminant
    V_AR   : AR_REC;
    V_R    : R_REC;
    AC_AR  : A_AR_REC;
    AC_R   : A_R_REC;
  end record;

X : DIS;
Y : DIS(5);

```

All of the following components depend on a discriminant of an unconstrained variable, and hence cannot be used as an actual in out generic parameter:

- components of variable X -- X.S, X.R, X.AS, X.AR;
- components of X.S and X.R -- X.S(l), X.R.D, X.R.A, X.R.V;
- components of variable X.V_AR(l) -- X.V_AR(l).V;
- components of variable X.V_R.E -- X.V_R.E.V (a subcomponent of variable X);
- components of variable X.AC_R.E -- X.AC_R.E.V (a subcomponent of variable X.AC_R.all).

In particular, note that X.S(1) is illegal as an actual parameter even though since the value of L must be at least 1, the referenced component always exists. Similarly, X.R.D and X.R.A are illegal as actual parameters. Note also that, except for V_AR(l).V and V_R.E.V, any of the above components of variable Y are legal, since Y is constrained. Finally, since objects of an access type do not have components, X.AS(1), X.AR.D, and X.AR.A are all legal, even though X.AS and X.AR are illegal. In addition, since any designated object is constrained (RM 3.7.2/10), it is always legal to pass a component of a designated object as an actual in out parameter. However, subcomponents of designated objects can be subject to the rule. For example, X.AC_R.E is a component of a designated object. It is also an unconstrained variable: hence, X.AC_R.E.V is subject to the rule.

S4. The following are the only legal subcomponents of X that can appear as an actual generic in out parameter: X.L (the discriminant), X.RC (a component), X.RC.D, X.RC.A, and X.RC.V (subcomponents), X.ARU, X.ARU.D, X.ARU.A, X.ARU.V, X.V_AR, X.V_AR(l), X.V_R, X.V_R.E, X.V_R.E.D, and X.V_R.E.A. In particular, note that X.V_R.E is an unconstrained variable having some components that depend on a discriminant, as is X.V_AR(l).

S5. The restriction regarding subcomponents that depend on discriminants is more severe for variables that are formal generic parameters. Such variables are always of mode in out (otherwise, the formal object would not be a variable). But it is irrelevant whether the discriminants of the formal object have default values or not:


```

type BUFFER (SIZE : NATURAL) is
  record
    POS : NATURAL := 0;
    VAL : STRING (1..SIZE);
  end record;

subtype BUFF_200 is BUFFER (200);
generic
  X : in out BUFF_200;           -- constrained
package P is
  generic
    Y : in out CHARACTER;
  package Q is end Q;

  RX : BUFF_200 renames X;
  package NQ is new Q (X.VAL(103)); -- illegal
  package NQ2 is new Q (RX.VAL(103)); -- illegal
end P;

Z : BUFFER(100) := ...;

package NP is new P (Z);        -- legal if P is legal

```

X is a variable having subcomponents that depend on a discriminant, namely, X.VAL and X.VAL(I). Since X is a formal In out parameter, these subcomponents are not allowed as actual In out parameters, and so, the instantiation of NQ is illegal. The constraint applicable to X is immaterial. Similarly, since RX renames X, RX denotes a formal In out parameter, and so, the instantiation of NQ2 is also illegal.

S6. Suppose we removed the illegal instantiations from P. Then the instantiation for NP will be meaningful and legal. In addition, we could then write:

```

use NP;
package NQ3 is new NP.Q (RX.VAL(103));

```

This instantiation is not illegal because NP.RX denotes Z, and Z is a constrained variable. However, NP.RX.VAL(103) will raise CONSTRAINT_ERROR, since 103 exceeds Z.VAL'LAST.

S7. Finally, suppose Q.Y were an In generic formal parameter. Then the instantiations for NQ and NQ2 would be legal. However, the instantiation for NP would raise CONSTRAINT_ERROR when the instantiated unit, NP, is elaborated; 103 exceeds the upper bound of the variable denoted by X (i.e., Z).

S8. Note the difference for subprogram In out parameters:

```

procedure PROC (X : in out BUFFER) is
  package NQ4 is new NP.Q (X.VAL(103));

```

This instantiation is legal since BUFFER, the type of PROC.X, does not have default discriminants. The instantiation might, of course, raise CONSTRAINT_ERROR. If BUFFER's discriminant had a default expression, then the instantiation for NQ4 would be illegal.

S9. The value of a generic In parameter is constant throughout the existence of an instantiation. In particular, for instantiations of generic subprograms, an implementation must ensure that every invocation of the subprogram uses the In parameter value provided at an instantiation. For example:

```

generic
  type T is private;
  DEF : T;
procedure P (X : T := DEF);

```

If the generic unit P is declared as a library unit or declared inside a library package, P can be instantiated as a library unit:

```

with Q;
procedure LIB_P is new P (INTEGER, Q.Y);

```

Every invocation of LIB_P must use the value provided by Q.Y at the time of LIB_P's instantiation.

Changes from July 1982

S10. An actual *In out* parameter must not be a formal parameter of mode *out* or a subcomponent thereof.

Changes from July 1980

S11. There are no significant changes.

Legality Rules

- L1. Formal and actual object parameters must have the same base type (RM 12.3.1/1 and RM 12.3.1/2).
- L2. An actual *In out* parameter must be a variable (RM 12.3.1/2).
- L3. An actual *In out* parameter must not be a parameter of mode *out* or a subcomponent thereof (RM 12.3.1/2).
- L4. If an actual *In out* parameter is a subcomponent of:
 - a variable declared by an object declaration, a record component declaration, or an array component declaration, and the variable has an unconstrained record type with default discriminants (RM 12.3.1/2 and RM 8.5/5); or
 - an *In out* subprogram or an entry formal parameter declared as an unconstrained record type with default discriminants; or
 - a generic formal *In out* parameter whose base type is a record type that has discriminants (with or without defaults),

then the subcomponent is not allowed as an actual *In out* generic parameter if the subcomponent depends on a discriminant (see RM 3.7.1/6-7) of the variable, subprogram formal parameter, or generic formal parameter, respectively.

- L5. If a generic formal *In* parameter has an unconstrained array type, then the actual parameter must not be an aggregate with an *others* choice (RM 4.3.2/5).
- L6. If a generic formal *In* parameter has a constrained array type, the actual parameter must not contain an association with an *others* choice together with other named associations (RM 4.3.2/5).

Exception Conditions

- E1. **CONSTRAINT_ERROR** is raised for a formal *In* parameter having a scalar type if the actual parameter's value does not lie in the range of the formal parameter's constraint.

- E2. **CONSTRAINT_ERROR** is raised for a formal in parameter having a non-null constrained array type if the number of components for corresponding dimensions (of the formal and actual parameter) is not the same.
- E3. **CONSTRAINT_ERROR** is raised for a constrained formal in parameter with discriminants if corresponding discriminant values are not equal.
- E4. **CONSTRAINT_ERROR** is raised for a formal in parameter having a constrained access type if the actual parameter's value is not null:
 - the designated type is an array type, and any index bound of the designated object does not equal the corresponding bound specified for the formal parameter.
 - the designated type is a type with discriminants and any discriminant of the designated object does not equal the corresponding value specified for the formal parameter.

Test Objectives and Design Guidelines

- T1. Check that formal and actual parameters cannot have different base types.
Implementation Guideline: Check for both modes, and when the type is a formal generic type.
- T2. Check that a generic actual in out parameter must be a variable.
Implementation Guideline: Try the following kinds of actual parameters: a declared constant (including components of a constant); a generic in parameter; a subprogram in parameter; a loop parameter; a named number, a parenthesized variable; a function call returning a result of a scalar, array, record, access, and private type; a sliced function result; a component of a composite function result; an attribute; an aggregate of variables; a qualified variable; an allocator; a record discriminant; an expression with an operator; and a variable as the operand of a type conversion.

Check that an actual in out parameter cannot be a subcomponent of an out subprogram or entry parameter.
- T3. Check that an actual in out parameter cannot be a subcomponent of a variable that is a record component, an array component, or an object declared with an object declaration, if the variable has an unconstrained type with default discriminants, and the subcomponent depends on a discriminant, i.e., the subcomponent is:
 - a component of the variable and the component:
 - is in a variant part of the variable;
 - is an array with at least one bound specified by a discriminant of the variable;
 - has a constrained record or private type with at least one discriminant specified by a discriminant of the variable;
 - has an access type with an explicit index or discriminant constraint using a discriminant of the variable;
 - a subcomponent of one of the above components.
- T4. Check that an actual in out parameter cannot be a subcomponent of an unconstrained subprogram or entry in out formal parameter if the subprogram or entry parameter has a record type with default discriminants and the subcomponent depends on a discriminant of the parameter's type (see the definition in T3).
- T5. Check that a subcomponent of a generic in out formal parameter, F, cannot be used as an

actual generic in out parameter if F's base type is a record type with discriminants (the existence of defaults is irrelevant), and the subcomponent depends on one of F's discriminants (see the definition in T3).

Implementation Guideline: Be sure to check the case when the generic formal variable, F, is specified with a constrained type mark and with a type that has discriminants without defaults.

Implementation Guideline: Note that F cannot have a formal generic type because no subcomponent accessible within the generic template can be dependent on a discriminant; at most, the only visible subcomponents of F are its discriminants.

T6. Check that an actual generic in out parameter can be:

- any subcomponent that does not depend on a discriminant, even if the enclosing variable is unconstrained;

Implementation Guideline: The variable should be a component of a record and array, a subprogram parameter (of mode in out), and a generic formal in out parameter.

- any subcomponent of an unconstrained variable of a record or private type if the discriminants of the variable do not have default values and the variable is not a formal generic in out parameter;

Implementation Guideline: Be sure to try subcomponents that depend on discriminants.

- any component of an object designated by an access value (even if the access variable depends on a discriminant);
- a slice of an array;
- a scalar variable;
- an access variable that does not depend on a discriminant.

Check that the formal parameter denotes the actual in an instantiation.

Implementation Guideline: Try some cases where the actual variable is unlikely to lie on an addressable machine boundary. Also try a task object.

Check that after instantiation, the subtype of an in out formal object is the same as that of the actual parameter, whether or not the formal parameter is constrained (see IG 12.1.1/T11).

T20. Check that the value of an in actual parameter is copied to the formal parameter.

Implementation Guideline: Use some cases where an implementation might use reference semantics for subprogram in parameters, e.g., when the parameter has a composite type.

Check that an in out formal parameter is not a copy of its actual parameter, but instead, denotes it directly.

T21. Check that an unconstrained in parameter having an array type or a type with discriminants has the constraints of the actual parameter.

T23. Check that default expressions for in parameters are only evaluated if there is no actual parameter.

Check that default expressions are evaluated in the order given by the generic declaration (see also IG 12.3.6/T2).

Implementation Guideline: Include a case where the default expression uses values of preceding formal parameters, some of whose values are provided by actual parameters, and some by the default expressions.

Check that an actual parameter must be provided if there is no default for the formal parameter.

T25. Check that CONSTRAINT_ERROR is raised for an in parameter having a scalar type if and only if the value of the actual parameter lies outside the range of the formal parameter.

Implementation Guideline: Check for integer, enumeration, float, and fixed types.

Implementation Guideline: Check both bounds of the range.

Implementation Guideline: Check when the `In` parameter has a formal generic type as well as a nonformal type.

- T26. For an `In` parameter having a non-null constrained array type, check that `CONSTRAINT_ERROR` is raised if and only if the actual parameter does not have the same number of components (per dimension) as the formal parameter.

Implementation Guideline: In particular, check that the bounds of the formal and actual parameters need not be the same.

Implementation Guideline: Some actual parameters should be strings and array slices.

Implementation Guideline: The `In` parameter should have a formal generic type as well as a nonformal type.

Check that `CONSTRAINT_ERROR` is not raised if both formal and actual parameters are null.

- T27. For a constrained `In` formal parameter having a record or private type with discriminants, check that `CONSTRAINT_ERROR` is raised if and only if corresponding discriminants of the actual and formal parameter do not have the same values.

Implementation Guideline: Use both formal and nonformal types.

- T28. For a constrained `In` formal parameter having an access type, check that `CONSTRAINT_ERROR` is raised if and only if the actual parameter is not null and the object designated by the actual parameter does not satisfy the formal parameter's constraint.

Implementation Guideline: Some actual parameters in the tests should have the value null. Actual parameters having a constrained and unconstrained access type should both be used.

12.3.2 Matching Rules for Formal Private Types

Semantic Ramifications

S1. When a generic formal type is a private type without discriminants, the corresponding actual parameter can be an unconstrained array type or an unconstrained type with discriminants. An instantiation with such an unconstrained type is illegal if the formal parameter is used in a context that requires a constrained type (RM 12.3.2/4), e.g., in an object declaration:

```
generic
  type FT is private;
package P is
  X : FT;                                -- potentially illegal
end P;

type TEXT (D : INTEGER := 0) is ...;

package PP is new P (STRING);            -- illegal
package QQ is new P (TEXT);              -- illegal
```

The first instantiation is illegal because `PP`'s declaration of `X` is illegal when `FT` is an unconstrained array type (see RM 3.6.1/6). The second instantiation is illegal because the declaration of `X` requires a type with default discriminants, if the type is unconstrained (RM 3.7.2/8). The fact that the actual parameter does have default discriminants doesn't matter since RM 12.3.2/4 says "The actual subtype must not be an unconstrained array type or an unconstrained type with discriminants," if the formal type is used in certain ways within the generic template. The rule applies equally to types with and without default discriminants.

The prohibition regarding the use of FT applies to types derived from FT (directly or indirectly) and to any name that denotes FT or a type derived from FT. For example, each T_i below is considered equivalent to FT for purposes of this rule:

```

subtype T1 is FT;
type T2 is new T1;
type T3 is new T2;
subtype T4 is T2;
type T5 is new T4;

```

S2. A name denoting a formal type can also be obtained by a generic instantiation:

```

generic
  type GT is private;
package R is
  subtype SGT is GT;
end R;

package NR is new R (FT);

```

NR.SGT is a name denoting FT.

S3. The following is a list of all contexts in which it is illegal to use T, an unconstrained array type or an unconstrained type with discriminants:

1. as the parent type in a derived type definition appearing in the full declaration of a private type (see RM 7.4.1/3), e.g.,

```

      type U is private;
private
      type U is new T;           -- potentially illegal

```

2. (if T's discriminants do not have defaults), as the subtype indication:

- a. in an object declaration declaring a variable (RM 3.6.1/6 and RM 3.7.2/8) (use in a constant declaration is okay (RM 3.6.1/7 and RM 3.7.2/9));
- b. in a record component declaration (RM 3.6.1/6 and RM 3.7.2/8);
- c. in an array component declaration (even as a generic formal parameter) (RM 3.6.1/6 and RM 3.7.2/8);
- d. in an allocator (e.g., new T) (see RM 3.6.1/8, RM 3.7.2/10, and RM 4.8/4);

3. as an actual generic parameter when the formal parameter is a private or limited private type (without discriminants), and the formal type parameter is used in one of these three contexts.

S4. The following example illustrates the potentially illegal contexts in which a formal generic private type can be used. The presence of any of these potentially illegal constructs makes any instantiation illegal if the actual type is an unconstrained array type or an unconstrained type with discriminants (whether or not the discriminants have defaults):

```

generic
  type FT is private;
  C : FT;
  CU: in out FT;           -- ok

```

```

    type DIS is range <>;
    type D is array (DIS) of FT;  -- potentially illegal (2c)
package P is
    type AFT is access FT;
    Y : AFT := new FT;           -- potentially illegal (2d)
    X : FT;                      -- potentially illegal (2a)
    type U is array (1..5) of FT; -- potentially illegal (2c)
    type V is
        record
            A : FT;              -- potentially illegal (2b)
        end record;
    type DER_FT is new FT;
    DER_X : DER_FT;             -- potentially illegal (2a)

    type INCO;
    type PRIV is private;

    generic
        type NEW_T is private;
        Y : NEW_T;
    package S is
        X : NEW_T;              -- potentially illegal (2a)
    end S;

    package S1 is new S (FT, C); -- potentially illegal (3)
    type INCO is new FT;        -- ok
    Z : INCO;                   -- potentially illegal (2a)

private
    type PRIV is new FT;        -- potentially illegal (1)

end P;
```

An instantiation of package P with an unconstrained array type (for example) would also be illegal if the potentially illegal declarations appeared in the package body or in some subunit of the package body.

s5. Since a generic body need not appear before an instantiation of the unit, it is not always possible to decide whether an instantiation is legal without considering the complete generic body and its subunits, if any:

```

generic
    type FT is private;
package P is
    X : INTEGER;
end P;

procedure Q is
    package PP is new P (STRING); -- potentially illegal
begin
    ...
end Q;
```

```

package body P is
  Y : FT;

end P;

```

```

-- now we see the instantiation
--   of P was illegal

```

In essence, the instantiation of P is discovered to be illegal only after P's body is processed. Note that P's body is itself quite legal. If the instantiation had been

```

package PP is new P (INTEGER);

```

nothing would be illegal in the above example. (These conclusions are unchanged if Q is turned into a package specification, although if Q is a package and the instantiation uses INTEGER instead of STRING, the instantiation will raise PROGRAM_ERROR (see IG 3.9/S).)

S6. Now suppose package P, procedure Q, and package body P are compiled separately. (To do so, we would have to insert "with P;" before Q's declaration.) If these units are compiled in the given order, the instantiation of P will not be illegal when Q is compiled since there are no potentially illegal uses of FT in P's package specification, and it is not illegal to compile Q before P's body (if any) has been compiled. When P's body is compiled, however, it will be discovered that the instantiation of P in Q is illegal, because of Y's declaration. Since Q has already been compiled, P's body can be rejected as unsuitable given the instantiations of P that are already known to exist. (Note that RM 10.3/3 says a compilation is not successful if any error is detected; 10.3/3 does not require that the detected error reside in the unit being compiled. Note also that "not successful" does not imply "illegal"; see RM 10.3/S.)

S7. Another possibility is to place the body for unit P in the library. In this case, the existence of an illegal instantiation in Q would only be indicated when attempting to execute a main program that uses Q. This treatment is justified by RM 1.6/3, which requires that illegal constructs be detected at compile time, but "compile time" is the time prior to an attempt to execute the main program.

S8. RM 10.3/9 allows an implementation to require that generic bodies (and subunits) be compiled together in a single compilation file. If an implementation imposes this requirement, it is still possible for a generic body to be compiled after an instantiation:

```

generic
  type T is private;
package P is
  X : INTEGER;
end P;

with P;
procedure Q is
  package NP is new P (INTEGER);
begin
  ...
end Q;

package body P is
  Y : T;
end P;

```

It must be possible to execute the main program, Q, without recompiling Q or any other unit. Although RM 10.3/8 allows an implementation to create additional dependences between units in a single compilation file, these additional dependences are for optimization purposes. If a

compilation file contains a set of units that can be executed as a main program in the absence of such "dependences," it must be possible to execute them as a main program even if the additional dependences are created. In the above example, this means Q is executable as a main program, even though a body has been provided after P's instantiation (i.e., the provision of the body does not make Q obsolete, so Q need not be recompiled). Of course, if the instantiation in Q has used an unconstrained array type, e.g., STRING, then the instantiation is illegal, and must be detected prior to execution of the main program.

S9. When a task type or a limited type with a task subcomponent is an actual parameter in an instantiation, then a dependent task (see IG 9.4/S) can be created within the instantiated unit. Hence, the code associated with such an instantiation is likely to be quite different from the code needed when instantiating with a type that is not a task, or that has no task subcomponents.

S10. The check that discriminants of a formal and actual type have the same subtype can only be suppressed by naming the actual parameter or generic unit in a SUPPRESS pragma; the formal parameter cannot be named since a SUPPRESS pragma cannot be given in a generic formal part (RM 11.7/3), and if the pragma is given after the occurrence of the formal parameter, it cannot have an effect on the checks performed during an instantiation, since only checks occurring after the pragma can be suppressed.

Changes from July 1982

S11. The legality of an instantiation using a type with discriminants is independent of whether the discriminants have defaults.

Changes from July 1980

S12. The actual subtype corresponding to a formal type with discriminants must be unconstrained.

S13. Corresponding discriminants of a formal and actual parameter must have the same base type, but need not have the same subtype.

Legality Rules

- L1. If a generic formal parameter is a nonlimited private type, the corresponding actual parameter must be a limited type (RM 12.3.2/2).
- L2. If a formal type parameter has a discriminant part, the corresponding actual parameter must be an unconstrained type with the same number of discriminants, and corresponding discriminants must have the same base type (RM 12.3.2/3).
- L3. Let FT be the name of a formal private or limited private type without discriminants, and let T be the name of a type denoting either FT or a type indirectly (or directly) derived from FT. (Note: a type is directly derived from its parent type. A type U is indirectly derived from a type W if U is directly derived from W or from a type indirectly derived from W.)

If the actual parameter corresponding to FT is an unconstrained array type or an unconstrained record, private, or limited private type with discriminants, then the instantiation for FT is illegal if the generic unit being instantiated contains (RM 12.3.2/4):

- an allocator of the form new T (RM 3.6.1/8, RM 3.7.2/10, and RM 4.8/4);
- a full declaration of a private type when the full declaration declares a derived type whose parent type is T (RM 7.4.1/3);
- a variable of type T declared in an object declaration (RM 3.6.1/6 and RM 3.7.2/8);
- a record component of type T (RM 3.7.2/8);

- an array component of type T (RM 3.6.1/6);
- an instantiation with actual parameter T if the corresponding formal parameter is used in one of these contexts (RM 12.3.2/4).

L4. An actual type parameter must have the form of a type mark (RM 12.3/2).

Exception Conditions

E1. For a formal parameter with discriminants, **CONSTRAINT_ERROR** is raised if corresponding discriminants do not have the same values for their range constraints.

Test Objectives and Design Guidelines

T1. If a generic formal parameter is a nonlimited private type, check that the actual parameter must not be:

- a limited private type;
- a type with subcomponents of a limited private type;
- a task type; or
- a type with subcomponents of a task type.

Implementation Guideline: Include a case where the actual parameter is a variant that does not have subcomponents violating the above rule, even though the base type does have such components.

Implementation Guideline: Include a case where the actual parameter is a formal limited private type of an enclosing generic unit.

T2. If a formal private or limited private type has a discriminant part, check that the actual parameter must:

- be an unconstrained type;
- have the same number of discriminants; and
- corresponding discriminants must have the same base types.

Implementation Guideline: Include some cases where the discriminants have a formal generic type.

Implementation Guideline: Include a case where the actual parameter is constrained, but otherwise satisfies the requirements.

T3. If a formal private or limited private type has a discriminant part, check that the actual parameter need not have identical discriminant names, and may have default discriminant values.

T4. Check that an instantiation is illegal if a formal parameter having a private or limited private type without discriminants is used as the parent type in a derived type definition appearing in a full declaration of a private type, and the actual type is an unconstrained array type or an unconstrained type with discriminants that do not have default values. (IG 7.4.1/T4 covers the case when the formal type is declared with discriminants.)

Implementation Guideline: Include types that are renamings of the formal type or that are derived directly or indirectly from the formal type.

Implementation Guideline: Include a case where the actual parameter is a formal parameter of an enclosing generic unit.

Implementation Guideline: Include tests where the instantiation appears after the generic body, before the generic body, and in a separately compiled unit that is compiled before or after the generic body.

Check that instantiations are not legal under the above circumstances if the actual parameter is an unconstrained type with default discriminant values.

- T5. Check that an instantiation is illegal if a formal parameter having a private or limited private type without discriminants is used:

- in an allocator;
- in an object declaration declaring a variable;
- in a record component declaration; or
- in an array type definition,

when the actual parameter is an unconstrained array type or an unconstrained type with discriminants that do or do not have default values.

Implementation Guideline: Follow the guidelines for IG 12.3.2/T4.

- T6. Check that an instantiation is legal if a formal private type (with or without discriminants) is used in a constant declaration and the actual parameter is a type with discriminants that do and do not have defaults.
- T7. Check that an instantiation is legal if a formal parameter having a limited private type without discriminants is used to declare an object or access type in a block that contains a selective wait with a terminate alternative, and the actual parameter's base type is either a task type or a type with a subcomponent of a task type.

Also check cases where the type used to declare the object or access type is a composite type with a subcomponent of the formal type.

- T8. Check that CONSTRAINT_ERROR is raised if the constraints of corresponding discriminants do not match.

Implementation Guideline: Some discriminants should have a nonformal type and some should have a formal type declared in the same or an enclosing generic formal part.

- T9. When an actual generic parameter is a task type or a type with a task subcomponent and the instantiated unit creates a dependent task, check that the dependent task is handled correctly (see IG 9.4/T10).

When an actual generic parameter is an access type whose designated type is a task type or a type with a component of a task type (and similarly, when the designated type is a formal type whose actual type is a task type or a type with a component of a task type), check that a created task is dependent on the unit that contains the actual access type declaration (see IG 9.4/T11).

- T10. Check that an actual type parameter cannot be a subtype indication with a constraint when the formal parameter is a private type without discriminants (see IG 12.3/T5).

- T20. Check that a discrete formal type denotes its actual parameter, and operations of the formal type are identified with corresponding operations of the actual type.

Implementation Guideline: To check this, for example, test that objects of a formal type that are declared in the visible part of a generic package can be used (after an instantiation) in expressions with objects declared with the actual type.

Implementation Guideline: In addition, declare a subtype of the formal type in a package. Derive a new type outside an instantiation of the package, using the instantiated subtype as the parent type. Check that the operations of the actual type are derived. In particular, check that operations not defined for the formal type inside the template are nonetheless derived for the actual type. (Note: "not defined" means the operation is either illegal for the formal type but not for the actual type or is given one meaning in the template (usually a predefined meaning) and a different meaning outside the instantiation.)

Implementation Guideline: The actual type should be an integer type, a character type, and some other enumeration type.

- T21. Repeat T20 for formal integer types.

- T22. Repeat T20 for formal floating point types.
- T23. Repeat T20 for formal fixed point types.
- T24. Repeat T20 for formal array types.
- T25. Repeat T20 for formal access types.
- T30. Repeat T20 for a formal private and limited private type when the actual parameter is an enumeration type.
- T31. Repeat T30 when the actual parameter is an integer type.
- T32. Repeat T30 when the actual parameter is a floating point type.
- T33. Repeat T30 when the actual parameter is a fixed point type.
- T34. Repeat T30 when the actual parameter is an array type.
- T35. Repeat T30 when the actual parameter is an access type.
- T36. Repeat T30 when the actual parameter is a type with discriminants.
- T40. Repeat T20 when the formal type has discriminants.

Implementation Guideline: Check that operations for selecting a formal's discriminant are properly associated with the corresponding operations for the actual parameter (see example in IG 12.1.2/S).

12.3.3 Matching Rules for Formal Scalar Types

Semantic Ramifications

- S1. Constraints that apply to an actual type parameter apply to the formal parameter after the instantiation, since the formal parameter denotes the actual parameter within the instantiated unit (RM 12.3/9).

Changes from July 1982

- S2. There are no changes.

Changes from July 1980

- S3. There are no significant changes.

Legality Rules

- L1. If a generic type definition has the form $\langle \rangle$, then the corresponding actual parameter must be an integer or enumeration type (RM 12.3.3/1).
- L2. If a generic type definition has the form $\text{range } \langle \rangle$, then the corresponding actual parameter must be an integer type (RM 12.3.3/1).
- L3. If a generic type definition has the form $\text{digits } \langle \rangle$, then the corresponding actual parameter must be a floating point type (RM 12.3.3/1).
- L4. If a generic type definition has the form $\text{delta } \langle \rangle$, then the corresponding actual parameter must be a fixed point type (RM 12.3.3/1).
- L5. A subtype indication containing an explicit range or accuracy constraint is not allowed as an actual type parameter (RM 12.3/2).

Test Objectives and Design Guidelines

- T1. Check that a generic actual type parameter must be an enumeration or integer type if the corresponding formal type definition has the form $\langle \rangle$.

Implementation Guideline: In particular, check that the instantiation is illegal if the actual parameter is a generic formal private type (of an enclosing generic unit).

- T2. Check that a generic actual type parameter must be an integer type if the corresponding formal type definition has the form `range <>`.

Implementation Guideline: In particular, check that the instantiation is illegal if the actual parameter is a generic formal private or discrete type.

- T3. Check that a generic actual type parameter must be a floating point type if the corresponding formal type definition has the form `digits <>`.

- T4. Check that a generic actual type parameter must be a fixed point type if the corresponding formal type definition has the form `delta <>`.

- T5. Check that for each form of formal scalar type, an instantiation uses the lower and upper bounds of the actual parameter within the instantiated unit.

Implementation Guideline: Check for initial values in declarations as well as in assignments (cf. IG 3.2.1/T11).

- T6. Check that a subtype indication with an explicit range or accuracy constraint is not allowed as an actual scalar type parameter (see IG 12.3/T5).

12.3.4 Matching Rules for Formal Array Types

Semantic Ramifications

- S1. Given a declaration of the following form:

```
generic
  type I is range <>;    -- index type
  type C is range <>;    -- component type
package P is
  generic
    type PI is range <>;
    type PC is range <>;
    type PA is array (PI range <>) of PC;
  package Q is end Q;

  type ARR is array (I range <>) of C;
  subtype I_5 is I range 1..5;      -- 1
  subtype C_10 is C range 5..10;    -- 2

  package NQ is new Q (I_5,
                      C_10,
                      ARR);        -- 3, 4
end P;
```

there are various possibilities for raising `CONSTRAINT_ERROR` when the enclosing package, P, is instantiated:

1. 1..5 is not compatible with I's actual range;
2. 5..10 is not compatible with C's actual range;
3. I'FIRST /= 1 or I'LAST /= 5;
4. C'FIRST /= 5 or C'LAST /= 10;

For the instantiation of NQ, Q.PA's index subtype is given by the actual parameter I_5, and

Q.PA's component subtype is given by the actual parameter C_10. This is why PA's index subtype has to have the bounds 1..5 and its component subtype, the bounds 5..10. It should also be noted that the following generic declaration is illegal because the instantiation contained within it is illegal:

```
generic
  type T is private;
package P is
  generic
    type ARR is array (NATURAL) of NATURAL;
  package Q is end Q;
  package NQ is new Q (T);  -- illegal
end P;
```

T is not an array type; it is a private type, and hence does not match Q's formal type.

S2. A formal array type cannot have an unconstrained component type when the actual parameter has a constrained component type:

```
type TEXT (L : INTEGER range 0..500 := 0) is
  record
    POS : INTEGER range 0..500 := 0;
    DATA : STRING (1..L);
  end record;

generic
  type I is range <>;
  type ARR is array (I) of TEXT;  -- unconstrained component type
package P is end P;

type ARR2 is array (1..50) of TEXT(50); -- constrained component type
subtype NAT_50 is NATURAL range 1..50;

package NP is new P (NAT_50, ARR2);  -- illegal
```

Although the base types of the formal and actual array components are the same, the instantiation is illegal because ARR2's component type is constrained and ARR's is not.

S3. When the component type is an access type, constraints can be imposed on objects designated by values of the access type. Constraints on the designated type can be imposed directly in an access type definition or indirectly on the access type itself:

```
type A_S is access STRING;
subtype A3_S is A_S (1..3);
type A_S3 is access STRING (1..3);
subtype ATHREE_S is A_S (1..THREE);  -- THREE is an INTEGER variable
subtype ONE_SIX is POSITIVE range 1..6;

generic
  type F1 is array (ONE_SIX) of ATHREE_S; -- constrained component
package P is end P;

type AR1_A_S is array (1..6) of A_S;
type AR2_A3_S is array (1..6) of A3_S;
type AR3_A_S3 is array (1..6) of A_S3;
```

```

package P1 is new P (AR1_A_S);      -- illegal
package P2 is new P (AR2_A3_S);
package P3 is new P (AR3_A_S3);    -- illegal

```

S4. The first instantiation is illegal because AR1_A_S's component type is an access type whose designated type is unconstrained, and F1's component type is an access type whose designated type is constrained.

The instantiation for P3 is illegal because the base type of AR3_A_S3's component is A_S3 and the base type of F1's component is A_S, and these are different base types.

The instantiation for P2 is legal because the base type of A3_S and ATHREE_S is A_S. Moreover, A3_S and ATHREE_S both impose a constraint on their designated types. CONSTRAINT_ERROR would be raised if the constraint values for A3_S and ATHREE_S were different, i.e., if THREE did not equal 3.

S5. Similar reasoning applies when the component type is a formal type:

```

generic
  type C is private;
  type F2 is array (ONE_SIX) of C;
package Q is end Q;

package Q1 is new Q (A_S, AR3_A_S3);    -- illegal
package Q2 is new Q (A3_S, AR1_A_S);    -- illegal
package Q3 is new Q (A_S, AR1_A_S);
package Q4 is new Q (ATHREE_S, AR2_A3_S);

```

The instantiation for Q1 is illegal because the base type of AR3_A_S3's component is A_S3, which is different from the base type A_S. The instantiation for Q2 is illegal because although AR1_A_S's component base type is the same as A3_S's base type, namely, A_S, AR1_A_S's component type is unconstrained, and A3_S is a constrained type. The instantiation of Q4 would raise CONSTRAINT_ERROR if THREE did not equal 3.

S6. An array type can be declared to have an unconstrained component type with discriminants if the discriminants have defaults:

```

type REC (D : INTEGER := 0) is
  record
    C : INTEGER;
  end record;
type ARR is array (1..4) of REC;    -- unconstrained component type

```

An instantiation of generic unit Q (from the last example) will be illegal, however, if REC and ARR are used as actual parameters:

```

package Q5 is new Q (REC, ARR);      -- illegal

```

This instantiation is illegal because the component type in an array type declaration must either be constrained or must have default discriminants. This means that formal parameter C is used in a context that implies its actual parameter must not be an unconstrained type with discriminants (RM 12.3.2/4). It does not matter that such an actual type exists.

S7. The check that components of a formal and actual array type have the same subtype (and similarly, the checks for the bounds or index subtypes of corresponding dimensions) can only be suppressed by naming the actual parameter or generic unit in a SUPPRESS pragma; the formal parameter cannot be named since a SUPPRESS pragma cannot be given in a generic formal part (RM 11.7/3). And, if the pragma is given after the occurrence of the formal parameter, it

cannot have an effect on the checks performed during an instantiation, since only checks occurring after the pragma can be suppressed.

Changes from July 1982

S8. There are no significant changes.

Changes from July 1980

S9. An instantiation is illegal if the component subtypes for the formal and actual parameters are not both constrained or are not both unconstrained.

Legality Rules

- L1. An actual parameter corresponding to a formal array type parameter must be an array type with the same number of indices (i.e., the same number of dimensions); the corresponding index positions must have the same base type; and the component base types must be the same (RM 12.3.4/2-4).
- L2. An actual array type parameter must be constrained if and only if the corresponding formal array parameter is constrained (RM 12.3.4/2).
- L3. A nonscalar component type of an actual array type parameter must be constrained if and only if the component type of the formal parameter is constrained (RM 12.3.4/4).
- L4. A subtype indication containing an explicit index constraint is not allowed as an actual type parameter (RM 12.3/2).

Exception Conditions

E1. CONSTRAINT_ERROR is raised if:

- both component types are constrained, but the constraint values are not equal.
- both array types are unconstrained and for some index position, the index subtype constraints of the formal and actual parameter are not equal.
- both array types are constrained, and for some index position, the bounds of the formal and actual parameter are not equal.

Test Objectives and Design Guidelines

- T1. Check that if a generic formal type is an array type, the actual parameter must also be an array type.
Implementation Guideline: Include a case where the actual parameter is a nonarray formal parameter of an enclosing generic unit. In particular, include a case where the actual parameter is a generic private type.
- T2. Check that a formal and actual generic array type must have the same number of dimensions.
Implementation Guideline: Include both constrained and unconstrained array types.
Implementation Guideline: Include a case where the actual parameter is a formal parameter of an enclosing generic unit.
- T3. Check that when the formal parameter is either constrained or unconstrained, corresponding index positions must have the same base type.
Implementation Guideline: Include cases where the index of the formal parameter type is a formal generic parameter of the same or an enclosing generic formal part, and the actual parameter is either a nongeneric type, or is a formal parameter of an enclosing unit.
- T4. Check that the component base types of corresponding formal and actual generic array parameters must be the same.

Implementation Guideline: Include all classes of base types in this test, and check that derived and parent types are not considered the same. Be sure to include access to access types as component types.

Implementation Guideline: Do the checks for the following cases:

- the component types are not themselves generic formal types;
- the component type of the formal parameter is a formal generic type declared earlier in the same generic formal part;
- the formal's component type is a formal type declared in an enclosing generic unit.
- the actual's component type is a formal type declared in an enclosing generic unit and the formal's component type is declared earlier in the generic formal part where the array type is declared.

T5. Check that if the component type is:

- a. a private, limited private, or record type with discriminants, or
- b. an access type designating objects of an array type or of a type with discriminants,

the formal and actual component types must both be either constrained or unconstrained.

Implementation Guideline: Include cases with and without default discriminants, and cases where the formal's component type is itself a formal type declared earlier in the same generic formal part; in this case, the actual parameter must always be constrained, even if the actual parameter has default discriminants. The actual parameter corresponding to the component type should sometimes be a formal type declared in an enclosing unit.

T6. Check that CONSTRAINT_ERROR is raised if an unconstrained formal and actual array type do not have index subtypes with the same bounds.

Implementation Guideline: Include a case where the index subtype is a formal type declared earlier in the generic formal part and the actual type is a formal parameter of an enclosing generic unit.

T7. Check that CONSTRAINT_ERROR is raised if the constraints imposed on a formal array parameter's component type do not equal those of the actual parameter.

Implementation Guideline: Check for all classes of component types that can have constraints, namely, enumeration, integer, float, fixed, array, record, private, and access types (two cases: with index constraints and with discriminant constraints).

Implementation Guideline: Include a case where the formal's component type and the actual component type are the same formal type.

T8. Check that CONSTRAINT_ERROR is raised if the index constraints for a constrained formal array parameter do not equal the constraints imposed on the actual parameter.

Implementation Guideline: Include a case where the formal's index constraint and the actual's index constraint are the same formal type.

T9. Check that a formal array type behaves correctly as an array type after an instantiation that is legal and that does not raise CONSTRAINT_ERROR. (This objective is satisfied by various tests of array types in other sections of the IG.)

T10. Check that an actual array type parameter cannot have an explicit index constraint (see IG 12.3/T5).

12.3.5 Matching Rules for Formal Access Types

Semantic Ramifications

S1. There are two ways of imposing constraints on objects designated by access values: 1) by imposing a constraint on an access type (yielding a constrained access type), and 2) by imposing a constraint in the access type's definition:

```

type A_S is access STRING;
subtype A3_S is A_S (1..3);      -- constrained access type
type A_S3 is access STRING (1..3); -- access to a constrained type

```

Now consider the following generic declaration:

```

generic
  type U is access STRING;
package PU is
  X : U := new STRING (1..3);
end PU;

```

and the following instantiations:

```

package P1 is new P (A_S);
package P2 is new P (A3_S);      -- illegal
package P3 is new P (A_S3);      -- illegal

```

The first instantiation is legal because U's designated (base) type is STRING, and this is also A_S's designated type. STRING is also A3_S's and A_S3's designated type. However, A3_S imposes a constraint on its designated type, and U imposes no such constraint; the instantiation is therefore illegal; one designated type is constrained and the other is not. P3's instantiation is illegal for the same reason.

S2. Now consider the case where the formal generic type imposes a constraint:

```

THREE : INTEGER := 3;
subtype S3 is STRING (1..THREE);

generic
  type FA_S3 is access S3;
  L : INTEGER;
package PC is
  XC : FA_S3 := new STRING (1..L);
end PC;

```

and consider the following instantiations:

```

package PC1 is new PC (A_S, 3);      -- illegal
package PC2 is new PC (A3_S, 3);
package PC3 is new PC (A_S3, 3);

package PC4 is new PC (A3_S, 4);      -- CONSTRAINT_ERROR
package PC5 is new PC (A_S3, 4);      -- CONSTRAINT_ERROR

subtype A4_S is A_S (2..4);
type A_S4 is access STRING (2..4);

package PC6 is new PC (A4_S, 3);      -- CONSTRAINT_ERROR
package PC7 is new PC (A_S4, 3);      -- CONSTRAINT_ERROR

```

The declaration of PC1 is illegal because A_S's designated type is unconstrained and FA_S3's designated type is constrained. The declarations of PC2 and PC3 are legal: A3_S, A_S3, and FA_S3 all have the same designated type and all impose a constraint on their designated types. Moreover, no exception is raised because the lower and upper bounds of the imposed constraints have the same values. Finally, no exception is raised by the allocator since the

value of L in the instantiated package satisfies the constraint imposed on FA_S3's designated type.

s3. The declarations of PC4, PC5, PC6, and PC7 are all legal because FA_S3, A3_S, A_S3, A4_S, and A_S4 all have the same designated type and all impose constraints on the designated type. They all raise CONSTRAINT_ERROR, but for different reasons. The instantiations for PC4 and PC5 raise CONSTRAINT_ERROR because the value of L in the allocator for XC does not satisfy the constraint imposed on FA_S3's designated type. The instantiations for PC6 and PC7 raise CONSTRAINT_ERROR because the values in the constraint imposed on FA_S3's designated type are not the same as the values imposed on A4_S's and A_S4's designated types.

s4. Now consider the case where the designated type is itself an access type:

```
type A_A3_S is access A3_S;
type A_A_S3 is access A_S (1..3);
type A_A_S is access A_S;
type A_A_S4 is access A_S4;
type A_A4_S is access A_S (2..4);
```

Let us consider the following generic unit and instantiations:

```
generic
  type V is access A_S;          -- access access STRING
package Q is
  X : V := new A_S' (new STRING (1..3));
end Q;

package Q1 is new Q (A_A_S);
package Q2 is new Q (A_A_S4);    -- illegal
package Q3 is new Q (A_A3_S);    -- illegal
package Q4 is new Q (A_A_S3);    -- illegal
```

The instantiation of Q1 is clearly legal, since A_S is V's designated type and that is also A_A_S's designated type. Q2's declaration is illegal because the designated types are different: A_S4's designated type is A_S4, and A_S4 is not the same as any other access type, since A_S4 is declared with a type declaration. It is immaterial that both A_S4 and A_S designate objects of type STRING.

Q3's declaration is illegal even though A_A3_S and V have the same designated base type (i.e., A_S). The declaration is illegal because V's designated type is unconstrained and A_A3_S's designated type is constrained. The same reasoning explains why Q4's declaration is illegal, since A3_S is just another name for A_S (1..3).

s5. Finally, let us consider when the formal type is an access to a constrained type:

```
generic
  type VC is access A3_S;
package QC is
  X : VC := new A_S' (new STRING (1..3));
end QC;
```

Now consider the following instantiations:

```
package QC1 is new QC (A_A3_S);
package QC2 is new QC (A_A_S3);
package QC3 is new QC (A_A_S);    -- illegal
```

```

package QC4 is new QC (A_A_S4);      -- illegal
package QC5 is new QC (A_A4_S);      -- CONSTRAINT_ERROR
package QC6 is new QC (A_A_S4);      -- CONSTRAINT_ERROR

```

The first two instantiations are legal because VC's designated type is A3_S'BASE (i.e., A_S) and this is the same as A_A3_S's and A_A_S3's designated type. Moreover, VC's designated type is constrained, and so is A_A3_S's and A_A_S3's designated type. Finally, no exception is raised by the instantiation, since the index constraint applied to VC's designated type specifies the same index values as the index constraint applied to A_A3_S's and A_A_S3's designated type.

QC3's instantiation is illegal since no constraint is imposed on A_A_S's designated type, and since VC's designated type is constrained.

QC4's instantiation is illegal since A_A_S4's designated type is A_S4'BASE = A_S4, but VC's designated base type is A3_S'BASE, i.e., A_S, and these are not the same base types.

s6. The instantiations of QC5 and QC6 raise CONSTRAINT_ERROR because the values of the constraint imposed on VC's designated type do not equal the values of the constraint imposed on A_A4_S's and A_A_S4's designated type.

s7. The check that components of a formal and actual access type impose the same constraints on the designated objects can only be suppressed by naming the actual parameter or the generic unit in a SUPPRESS pragma; the formal parameter cannot be named since a SUPPRESS pragma cannot be given in a generic formal part (RM 11.7/3), and if the pragma is given after the occurrence of the formal parameter, it cannot have an effect on the checks performed during an instantiation, since only checks occurring after the pragma can be suppressed.

Changes from July 1982

s8. There are no significant changes.

Changes from July 1980

s9. For the formal and actual parameter, the designated subtypes must either both be constrained or both unconstrained.

Legality Rules

- L1. If a formal generic parameter is an access type, its corresponding actual parameter must also be an access type, and the formal and actual parameters must have the same designated (base) type (RM 12.3.5/1).
- L2. A designated nonscalar type of an actual access type parameter must be constrained and only if the designated type of the corresponding formal parameter is constrained (RM 12.3.5/1).
- L3. A subtype indication containing an explicit index or discriminant constraint is not allowed as an actual type parameter (RM 12.3/2).

Exception Conditions

- E1. CONSTRAINT_ERROR is raised if the designated type of a formal access subtype is constrained and the constraint values of the formal's designated subtype do not equal those of the actual's designated subtype.

Test Objectives and Design Guidelines

In the following objectives, FT is a formal access type declared by type FT is access T. The actual parameter corresponding to FT is AU, declared as type AU is access U.

- T1. Check that if a formal generic type is an access type, the actual generic type must also be an access type.

Implementation Guideline: Check that FT does not match an actual parameter, W, when W is the same as FT's designated type T, and when T and W are not generic formal types. (For example, W might be STRING and FT might be access STRING.) Check for all type classes: enumeration, integer, float, fixed, array (vary the index and component types; constrained and not constrained), record (with and without discriminants; constrained and not constrained), private (with and without discriminants; constrained and not constrained), and limited private.

Implementation Guideline: Repeat the above check when T is a formal generic type declared earlier in the same generic formal part as FT, and W is either a nonformal type or is a formal type declared in an enclosing generic unit.

Implementation Guideline: Repeat the above check when T and W are generic formal types appearing in an enclosing generic declaration.

- T2. If the formal and actual generic parameters are both access types, check that the designated (base) types must be the same.

Implementation Guideline: Check that FT does not match AU when both T and U are not formal generic types. Check for all type classes of T: enumeration, integer, float, fixed, array (vary the index and component types; constrained and not constrained), record (with and without discriminants; both T and U unconstrained, or the actual parameter being a constrained access type when the formal is an access-to-constrained type), access, private (with and without discriminants; constrained and not constrained), and limited private. Let T and U be derived types in some test cases.

Implementation Guideline: Repeat some of the above checks when T is a formal generic type declared earlier in the same generic formal part as FT.

Implementation Guideline: Repeat some of the above checks when T is a generic formal type appearing in an enclosing generic declaration.

Implementation Guideline: Repeat some of the above checks when U is a generic formal type appearing in an enclosing generic declaration, and T is either not a generic formal type or is a formal type declared earlier in the same generic formal part as FT.

Implementation Guideline: Repeat some of the above checks when both T and U are formal types appearing in an enclosing generic declaration. (Note that T might be declared as a discrete type (<>), and U as an integer type, range <>.)

- T3. If T is an array type or a type with discriminants and the base type of U is the same as T's base type, check that U must be constrained if and only if T is constrained.

Implementation Guideline: Note that the constraint associated with an actual parameter's designated objects can either be imposed directly in the access type's definition (e.g., type AU is access STRING(1..3)) or by constraining the access type (type A_S is access STRING; subtype AU is A_S(1..3)). Both forms of actual parameter constraint should be used in the test.

Implementation Guideline: Create tests for the following cases:

- T and U are not generic formal types;
- T is a formal type declared in the same generic formal part as FT;
- U is a formal type of an enclosing generic declaration; T is either not a generic formal type or is a formal type declared in the same generic formal part as FT;

Implementation Guideline: Repeat the above tests when T is itself an access type designating objects of an array type or a type with discriminants.

- T4. Check that CONSTRAINT_ERROR is raised when constraints imposed on T do not have the same values as constraints imposed on U.

Implementation Guideline: Check for all type classes: enumeration, integer, float, fixed, array, record, private, limited private, and access. Access types (at least) should be checked in a separate test.

Implementation Guideline: Create tests for the following cases:

- T and U are not generic formal types;
- T is a formal type declared in the same generic formal part as FT;
- U is a formal type of an enclosing generic declaration; T is either not a generic formal type or is a formal type declared in the same generic formal part as FT;

Implementation Guideline: When checking constraints of scalar types, T or U should be a base type in some cases.

- T5. Check that all access types may be passed as an actual type parameter, and that any constraints on the designated objects are enforced. (This test objective is satisfied by tests associated with access type constraint checks.)
- T6. Check that if T is a formal private or limited private type without discriminants, and an allocator of the form `new T` appears within the generic unit, then an instantiation is illegal if T's actual parameter is an unconstrained array type or a type with discriminants (see IG 12.3.2/T5).
- T7. Check that an actual access type parameter cannot have an explicit index or discriminant constraint (see IG 12.3/T5).

12.3.6 Matching Rules for Formal Subprograms

Semantic Ramifications

S1. The evaluation of a name is described in RM 4.1/10. As pointed out in IG 12.1.3/S, the time when a default subprogram name is evaluated only matters when the name denotes an entry of a task or a member of an entry family.

S2. When evaluating the expressions or the access values supplied in a default name, it is important to keep in mind the required order of evaluations in a generic instantiation: first the actual parameters are evaluated in an undefined order, then any needed default expressions or subprogram names are evaluated in the order of the formal parameter declarations (RM 12.3/17). Hence, in the following example,

```
generic
  I : INTEGER := 1;
  with procedure P is T.E(I);
```

the default entry denoted by P is determined either by I's actual parameter value or by the default value, 1.

S3. When a default subprogram is specified with a box, the corresponding default parameter is determined at the point of the instantiation, independently of whether the instantiation is ever elaborated:

```
generic
  type T is private;
package GP is
  generic
    with procedure PAR (X : T) is <>; -- (1)
  package Q is
    end Q;

  package body Q is ... end;

  procedure PAR (Y : INTEGER);           -- (2)
  procedure PAR (Z : T);                 -- (3)

  package NQ is new Q;                   -- uses (3)
end GP;
```

```

package body GP is ... end GP;

procedure PAR (X : INTEGER) is ... end;      -- (4)

package NP is new GP (INTEGER);

package NNQ is new NP.Q;                      -- uses (4)

```

This example illustrates that the default subprogram for GP.Q must be identified at the point where Q is instantiated. The initial declaration at (1) is legal even though there is no PAR procedure visible that has a parameter of type T. The instantiation for NQ uses the PAR procedure declared at (3) as the default actual parameter. If (3) were deleted, NQ's instantiation would be illegal, and so GP would be illegal. Finally, the instantiation for NNQ uses the PAR procedure declared at (4), since T is known to denote INTEGER within NP, and the only visible declaration of PAR (with an INTEGER parameter) is the one declared at (4). Note that within NP, NP.Q uses the procedure declared at (3) even though the two PAR procedures in NP have the same parameter profile. The identification of NQ's default parameter is fixed prior to any instantiation of GP.

S4. Although the parameter modes must be the same for matching subprograms, the modes are not strict, including whether a single match exists. (Note: there are two ways to have two visible subprograms with the same parameter type profile and different parameter modes: 1) via the package for different packages (see IG 8.4/S); and 2) via a generic package instantiation. Example 12.3.3.)

S5. When generic units have subprogram parameters, certain actual parameters can cause implementation difficulties. For example, most compilers will probably treat `"!="` in a special manner, since it must be the complement of `"="`. Even though `"!="` cannot be user-defined directly, it is indirectly defined when `"="` is defined. Passing a user-defined `"!="` operator may mean passing a user-defined `"="` function and complementing the result. Finally, subprogram parameters can be passed as generic parameters, although the pragma does not have to be obeyed in this case (RM 6.3.2/4).

S6. For object types, the decision about whether or not to invoke the function is determined by the actual and formal parameter modes. The function must be invoked if it is associated with an actual parameter and must not be invoked if it is associated with a formal subprogram parameter. Similarly, strings that are operator symbols are potentially ambiguous as actual parameters. If the formal parameter is an object, the string interpretation is used; if it is a subprogram parameter, the operator interpretation is used.

```

package P is
  type T1;
  type T2;
  type T3;
  type T4;
  type T5;
  type T6;
  type T7;
  type T8;
  type T9;
  type T10;
  type T11;
  type T12;
  type T13;
  type T14;
  type T15;
  type T16;
  type T17;
  type T18;
  type T19;
  type T20;
  type T21;
  type T22;
  type T23;
  type T24;
  type T25;
  type T26;
  type T27;
  type T28;
  type T29;
  type T30;
  type T31;
  type T32;
  type T33;
  type T34;
  type T35;
  type T36;
  type T37;
  type T38;
  type T39;
  type T40;
  type T41;
  type T42;
  type T43;
  type T44;
  type T45;
  type T46;
  type T47;
  type T48;
  type T49;
  type T50;
  type T51;
  type T52;
  type T53;
  type T54;
  type T55;
  type T56;
  type T57;
  type T58;
  type T59;
  type T60;
  type T61;
  type T62;
  type T63;
  type T64;
  type T65;
  type T66;
  type T67;
  type T68;
  type T69;
  type T70;
  type T71;
  type T72;
  type T73;
  type T74;
  type T75;
  type T76;
  type T77;
  type T78;
  type T79;
  type T80;
  type T81;
  type T82;
  type T83;
  type T84;
  type T85;
  type T86;
  type T87;
  type T88;
  type T89;
  type T90;
  type T91;
  type T92;
  type T93;
  type T94;
  type T95;
  type T96;
  type T97;
  type T98;
  type T99;
  type T100;
  type T101;
  type T102;
  type T103;
  type T104;
  type T105;
  type T106;
  type T107;
  type T108;
  type T109;
  type T110;
  type T111;
  type T112;
  type T113;
  type T114;
  type T115;
  type T116;
  type T117;
  type T118;
  type T119;
  type T120;
  type T121;
  type T122;
  type T123;
  type T124;
  type T125;
  type T126;
  type T127;
  type T128;
  type T129;
  type T130;
  type T131;
  type T132;
  type T133;
  type T134;
  type T135;
  type T136;
  type T137;
  type T138;
  type T139;
  type T140;
  type T141;
  type T142;
  type T143;
  type T144;
  type T145;
  type T146;
  type T147;
  type T148;
  type T149;
  type T150;
  type T151;
  type T152;
  type T153;
  type T154;
  type T155;
  type T156;
  type T157;
  type T158;
  type T159;
  type T160;
  type T161;
  type T162;
  type T163;
  type T164;
  type T165;
  type T166;
  type T167;
  type T168;
  type T169;
  type T170;
  type T171;
  type T172;
  type T173;
  type T174;
  type T175;
  type T176;
  type T177;
  type T178;
  type T179;
  type T180;
  type T181;
  type T182;
  type T183;
  type T184;
  type T185;
  type T186;
  type T187;
  type T188;
  type T189;
  type T190;
  type T191;
  type T192;
  type T193;
  type T194;
  type T195;
  type T196;
  type T197;
  type T198;
  type T199;
  type T200;
  type T201;
  type T202;
  type T203;
  type T204;
  type T205;
  type T206;
  type T207;
  type T208;
  type T209;
  type T210;
  type T211;
  type T212;
  type T213;
  type T214;
  type T215;
  type T216;
  type T217;
  type T218;
  type T219;
  type T220;
  type T221;
  type T222;
  type T223;
  type T224;
  type T225;
  type T226;
  type T227;
  type T228;
  type T229;
  type T230;
  type T231;
  type T232;
  type T233;
  type T234;
  type T235;
  type T236;
  type T237;
  type T238;
  type T239;
  type T240;
  type T241;
  type T242;
  type T243;
  type T244;
  type T245;
  type T246;
  type T247;
  type T248;
  type T249;
  type T250;
  type T251;
  type T252;
  type T253;
  type T254;
  type T255;
  type T256;
  type T257;
  type T258;
  type T259;
  type T260;
  type T261;
  type T262;
  type T263;
  type T264;
  type T265;
  type T266;
  type T267;
  type T268;
  type T269;
  type T270;
  type T271;
  type T272;
  type T273;
  type T274;
  type T275;
  type T276;
  type T277;
  type T278;
  type T279;
  type T280;
  type T281;
  type T282;
  type T283;
  type T284;
  type T285;
  type T286;
  type T287;
  type T288;
  type T289;
  type T290;
  type T291;
  type T292;
  type T293;
  type T294;
  type T295;
  type T296;
  type T297;
  type T298;
  type T299;
  type T300;
  type T301;
  type T302;
  type T303;
  type T304;
  type T305;
  type T306;
  type T307;
  type T308;
  type T309;
  type T310;
  type T311;
  type T312;
  type T313;
  type T314;
  type T315;
  type T316;
  type T317;
  type T318;
  type T319;
  type T320;
  type T321;
  type T322;
  type T323;
  type T324;
  type T325;
  type T326;
  type T327;
  type T328;
  type T329;
  type T330;
  type T331;
  type T332;
  type T333;
  type T334;
  type T335;
  type T336;
  type T337;
  type T338;
  type T339;
  type T340;
  type T341;
  type T342;
  type T343;
  type T344;
  type T345;
  type T346;
  type T347;
  type T348;
  type T349;
  type T350;
  type T351;
  type T352;
  type T353;
  type T354;
  type T355;
  type T356;
  type T357;
  type T358;
  type T359;
  type T360;
  type T361;
  type T362;
  type T363;
  type T364;
  type T365;
  type T366;
  type T367;
  type T368;
  type T369;
  type T370;
  type T371;
  type T372;
  type T373;
  type T374;
  type T375;
  type T376;
  type T377;
  type T378;
  type T379;
  type T380;
  type T381;
  type T382;
  type T383;
  type T384;
  type T385;
  type T386;
  type T387;
  type T388;
  type T389;
  type T390;
  type T391;
  type T392;
  type T393;
  type T394;
  type T395;
  type T396;
  type T397;
  type T398;
  type T399;
  type T400;
  type T401;
  type T402;
  type T403;
  type T404;
  type T405;
  type T406;
  type T407;
  type T408;
  type T409;
  type T410;
  type T411;
  type T412;
  type T413;
  type T414;
  type T415;
  type T416;
  type T417;
  type T418;
  type T419;
  type T420;
  type T421;
  type T422;
  type T423;
  type T424;
  type T425;
  type T426;
  type T427;
  type T428;
  type T429;
  type T430;
  type T431;
  type T432;
  type T433;
  type T434;
  type T435;
  type T436;
  type T437;
  type T438;
  type T439;
  type T440;
  type T441;
  type T442;
  type T443;
  type T444;
  type T445;
  type T446;
  type T447;
  type T448;
  type T449;
  type T450;
  type T451;
  type T452;
  type T453;
  type T454;
  type T455;
  type T456;
  type T457;
  type T458;
  type T459;
  type T460;
  type T461;
  type T462;
  type T463;
  type T464;
  type T465;
  type T466;
  type T467;
  type T468;
  type T469;
  type T470;
  type T471;
  type T472;
  type T473;
  type T474;
  type T475;
  type T476;
  type T477;
  type T478;
  type T479;
  type T480;
  type T481;
  type T482;
  type T483;
  type T484;
  type T485;
  type T486;
  type T487;
  type T488;
  type T489;
  type T490;
  type T491;
  type T492;
  type T493;
  type T494;
  type T495;
  type T496;
  type T497;
  type T498;
  type T499;
  type T500;
  type T501;
  type T502;
  type T503;
  type T504;
  type T505;
  type T506;
  type T507;
  type T508;
  type T509;
  type T510;
  type T511;
  type T512;
  type T513;
  type T514;
  type T515;
  type T516;
  type T517;
  type T518;
  type T519;
  type T520;
  type T521;
  type T522;
  type T523;
  type T524;
  type T525;
  type T526;
  type T527;
  type T528;
  type T529;
  type T530;
  type T531;
  type T532;
  type T533;
  type T534;
  type T535;
  type T536;
  type T537;
  type T538;
  type T539;
  type T540;
  type T541;
  type T542;
  type T543;
  type T544;
  type T545;
  type T546;
  type T547;
  type T548;
  type T549;
  type T550;
  type T551;
  type T552;
  type T553;
  type T554;
  type T555;
  type T556;
  type T557;
  type T558;
  type T559;
  type T560;
  type T561;
  type T562;
  type T563;
  type T564;
  type T565;
  type T566;
  type T567;
  type T568;
  type T569;
  type T570;
  type T571;
  type T572;
  type T573;
  type T574;
  type T575;
  type T576;
  type T577;
  type T578;
  type T579;
  type T580;
  type T581;
  type T582;
  type T583;
  type T584;
  type T585;
  type T586;
  type T587;
  type T588;
  type T589;
  type T590;
  type T591;
  type T592;
  type T593;
  type T594;
  type T595;
  type T596;
  type T597;
  type T598;
  type T599;
  type T600;
  type T601;
  type T602;
  type T603;
  type T604;
  type T605;
  type T606;
  type T607;
  type T608;
  type T609;
  type T610;
  type T611;
  type T612;
  type T613;
  type T614;
  type T615;
  type T616;
  type T617;
  type T618;
  type T619;
  type T620;
  type T621;
  type T622;
  type T623;
  type T624;
  type T625;
  type T626;
  type T627;
  type T628;
  type T629;
  type T630;
  type T631;
  type T632;
  type T633;
  type T634;
  type T635;
  type T636;
  type T637;
  type T638;
  type T639;
  type T640;
  type T641;
  type T642;
  type T643;
  type T644;
  type T645;
  type T646;
  type T647;
  type T648;
  type T649;
  type T650;
  type T651;
  type T652;
  type T653;
  type T654;
  type T655;
  type T656;
  type T657;
  type T658;
  type T659;
  type T660;
  type T661;
  type T662;
  type T663;
  type T664;
  type T665;
  type T666;
  type T667;
  type T668;
  type T669;
  type T670;
  type T671;
  type T672;
  type T673;
  type T674;
  type T675;
  type T676;
  type T677;
  type T678;
  type T679;
  type T680;
  type T681;
  type T682;
  type T683;
  type T684;
  type T685;
  type T686;
  type T687;
  type T688;
  type T689;
  type T690;
  type T691;
  type T692;
  type T693;
  type T694;
  type T695;
  type T696;
  type T697;
  type T698;
  type T699;
  type T700;
  type T701;
  type T702;
  type T703;
  type T704;
  type T705;
  type T706;
  type T707;
  type T708;
  type T709;
  type T710;
  type T711;
  type T712;
  type T713;
  type T714;
  type T715;
  type T716;
  type T717;
  type T718;
  type T719;
  type T720;
  type T721;
  type T722;
  type T723;
  type T724;
  type T725;
  type T726;
  type T727;
  type T728;
  type T729;
  type T730;
  type T731;
  type T732;
  type T733;
  type T734;
  type T735;
  type T736;
  type T737;
  type T738;
  type T739;
  type T740;
  type T741;
  type T742;
  type T743;
  type T744;
  type T745;
  type T746;
  type T747;
  type T748;
  type T749;
  type T750;
  type T751;
  type T752;
  type T753;
  type T754;
  type T755;
  type T756;
  type T757;
  type T758;
  type T759;
  type T760;
  type T761;
  type T762;
  type T763;
  type T764;
  type T765;
  type T766;
  type T767;
  type T768;
  type T769;
  type T770;
  type T771;
  type T772;
  type T773;
  type T774;
  type T775;
  type T776;
  type T777;
  type T778;
  type T779;
  type T780;
  type T781;
  type T782;
  type T783;
  type T784;
  type T785;
  type T786;
  type T787;
  type T788;
  type T789;
  type T790;
  type T791;
  type T792;
  type T793;
  type T794;
  type T795;
  type T796;
  type T797;
  type T798;
  type T799;
  type T800;
  type T801;
  type T802;
  type T803;
  type T804;
  type T805;
  type T806;
  type T807;
  type T808;
  type T809;
  type T810;
  type T811;
  type T812;
  type T813;
  type T814;
  type T815;
  type T816;
  type T817;
  type T818;
  type T819;
  type T820;
  type T821;
  type T822;
  type T823;
  type T824;
  type T825;
  type T826;
  type T827;
  type T828;
  type T829;
  type T830;
  type T831;
  type T832;
  type T833;
  type T834;
  type T835;
  type T836;
  type T837;
  type T838;
  type T839;
  type T840;
  type T841;
  type T842;
  type T843;
  type T844;
  type T845;
  type T846;
  type T847;
  type T848;
  type T849;
  type T850;
  type T851;
  type T852;
  type T853;
  type T854;
  type T855;
  type T856;
  type T857;
  type T858;
  type T859;
  type T860;
  type T861;
  type T862;
  type T863;
  type T864;
  type T865;
  type T866;
  type T867;
  type T868;
  type T869;
  type T870;
  type T871;
  type T872;
  type T873;
  type T874;
  type T875;
  type T876;
  type T877;
  type T878;
  type T879;
  type T880;
  type T881;
  type T882;
  type T883;
  type T884;
  type T885;
  type T886;
  type T887;
  type T888;
  type T889;
  type T890;
  type T891;
  type T892;
  type T893;
  type T894;
  type T895;
  type T896;
  type T897;
  type T898;
  type T899;
  type T900;
  type T901;
  type T902;
  type T903;
  type T904;
  type T905;
  type T906;
  type T907;
  type T908;
  type T909;
  type T910;
  type T911;
  type T912;
  type T913;
  type T914;
  type T915;
  type T916;
  type T917;
  type T918;
  type T919;
  type T920;
  type T921;
  type T922;
  type T923;
  type T924;
  type T925;
  type T926;
  type T927;
  type T928;
  type T929;
  type T930;
  type T931;
  type T932;
  type T933;
  type T934;
  type T935;
  type T936;
  type T937;
  type T938;
  type T939;
  type T940;
  type T941;
  type T942;
  type T943;
  type T944;
  type T945;
  type T946;
  type T947;
  type T948;
  type T949;
  type T950;
  type T951;
  type T952;
  type T953;
  type T954;
  type T955;
  type T956;
  type T957;
  type T958;
  type T959;
  type T960;
  type T961;
  type T962;
  type T963;
  type T964;
  type T965;
  type T966;
  type T967;
  type T968;
  type T969;
  type T970;
  type T971;
  type T972;
  type T973;
  type T974;
  type T975;
  type T976;
  type T977;
  type T978;
  type T979;
  type T980;
  type T981;
  type T982;
  type T983;
  type T984;
  type T985;
  type T986;
  type T987;
  type T988;
  type T989;
  type T990;
  type T991;
  type T992;
  type T993;
  type T994;
  type T995;
  type T996;
  type T997;
  type T998;
  type T999;
  type T1000;
  type T1001;
  type T1002;
  type T1003;
  type T1004;
  type T1005;
  type T1006;
  type T1007;
  type T1008;
  type T1009;
  type T1010;
  type T1011;
  type T1012;
  type T1013;
  type T1014;
  type T1015;
  type T1016;
  type T1017;
  type T1018;
  type T1019;
  type T1020;
  type T1021;
  type T1022;
  type T1023;
  type T1024;
  type T1025;
  type T1026;
  type T1027;
  type T1028;
  type T1029;
  type T1030;
  type T1031;
  type T1032;
  type T1033;
  type T1034;
  type T1035;
  type T1036;
  type T1037;
  type T1038;
  type T1039;
  type T1040;
  type T1041;
  type T1042;
  type T1043;
  type T1044;
  type T1045;
  type T1046;
  type T1047;
  type T1048;
  type T1049;
  type T1050;
  type T1051;
  type T1052;
  type T1053;
  type T1054;
  type T1055;
  type T1056;
  type T1057;
  type T1058;
  type T1059;
  type T1060;
  type T1061;
  type T1062;
  type T1063;
  type T1064;
  type T1065;
  type T1066;
  type T1067;
  type T1068;
  type T1069;
  type T1070;
  type T1071;
  type T1072;
  type T1073;
  type T1074;
  type T1075;
  type T1076;
  type T1077;
  type T1078;
  type T1079;
  type T1080;
  type T1081;
  type T1082;
  type T1083;
  type T1084;
  type T1085;
  type T1086;
  type T1087;
  type T1088;
  type T1089;
  type T1090;
  type T1091;
  type T1092;
  type T1093;
  type T1094;
  type T1095;
  type T1096;
  type T1097;
  type T1098;
  type T1099;
  type T1100;
  type T1101;
  type T1102;
  type T1103;
  type T1104;
  type T1105;
  type T1106;
  type T1107;
  type T1108;
  type T1109;
  type T1110;
  type T1111;
  type T1112;
  type T1113;
  type T1114;
  type T1115;
  type T1116;
  type T1117;
  type T1118;
  type T1119;
  type T1120;
  type T1121;
  type T1122;
  type T1123;
  type T1124;
  type T1125;
  type T1126;
  type T1127;
  type T1128;
  type T1129;
  type T1130;
  type T1131;
  type T1132;
  type T1133;
  type T1134;
  type T1135;
  type T1136;
  type T1137;
  type T1138;
  type T1139;
  type T1140;
  type T1141;
  type T1142;
  type T1143;
  type T1144;
  type T1145;
  type T1146;
  type T1147;
  type T1148;
  type T1149;
  type T1150;
  type T1151;
  type T1152;
  type T1153;
  type T1154;
  type T1155;
  type T1156;
  type T1157;
  type T1158;
  type T1159;
  type T1160;
  type T1161;
  type T1162;
  type T1163;
  type T1164;
  type T1165;
  type T1166;
  type T1167;
  type T1168;
  type T1169;
  type T1170;
  type T1171;
  type T1172;
  type T1173;
  type T1174;
  type T1175;
  type T1176;
  type T1177;
  type T1178;
  type T1179;
  type T1180;
  type T1181;
  type T1182;
  type T1183;
  type T1184;
  type T1185;
  type T1186;
  type T1187;
  type T1188;
  type T1189;
  type T1190;
  type T1191;
  type T1192;
  type T1193;
  type T1194;
  type T1195;
  type T1196;
  type T1197;
  type T1198;
  type T1199;
  type T1200;
  type T1201;
  type T1202;
  type T1203;
  type T1204;
  type T1205;
  type T1206;
  type T1207;
  type T1208;
  type T1209;
  type T1210;
  type T1211;
  type T1212;
  type T1213;
  type T1214;
  type T1215;
  type T1216;
  type T1217;
  type T1218;
  type T1219;
  type T1220;
  type T1221;
  type T1222;
  type T1223;
  type T1224;
  type T1225;
  type T1226;
  type T1227;
  type T1228;
  type T1229;
  type T1230;
  type T1231;
  type T1232;
  type T1233;
  type T1234;
  type T1235;
  type T1236;
  type T1237;
  type T1238;
  type T1239;
  type T1240;
  type T1241;
  type T1242;
  type T1243;
  type T1244;
  type T1245;
  type T1246;
  type T1247;
  type T1248;
  type T1249;
  type T1250;
  type T1251;
  type T1252;
  type T1253;
  type T1254;
  type T1255;
  type T1256;
  type T1257;
  type T1258;
  type T1259;
  type T1260;
  type T1261;
  type T1262;
  type T1263;
  type T1264;
  type T1265;
  type T1266;
  type T1267;
  type T1268;
  type T1269;
  type T1270;
  type T1271;
  type T1272;
  type T1273;
  type T1274;
  type T1275;
  type T1276;
  type T1277;
  type T1278;
  type T1279;
  type T1280;
  type T1281;
  type T1282;
  type T1283;
  type T1284;
  type T1285;
  type T1286;
  type T1287;
  type T1288;
  type T1289;
  type T1290;
  type T1291;
  type T1292;
  type T1293;
  type T1294;
  type T1295;
  type T1296;
  type T1297;
  type T1298;
  type T1299;
  type T1300;
  type T1301;
  type T1302;
  type T1303;
  type T1304;
  type T1305;
  type T1306;
  type T1307;
  type T1308;
  type T1309;
  type T1310;
  type T1311;
  type T1312;
  type T1313;
  type T1314;
  type T1315;
  type T1316;
  type T1317;
  type T1318;
  type T1319;
  type T1320;
  type T1321;
  type T1322;
  type T1323;
  type T1324;
  type T1325;
  type T1326;
  type T1327;
  type T1328;
  type T1329;
  type T1330;
  type T1331;
  type T1332;
  type T1333;
  type T1334;
  type T1335;
  type T1336;
  type T1337;
  type T1338;
  type T1339;
  type T1340;
  type T1341;
  type T1342;
  type T1343;
  type T1344;
  type T1345;
  type T1346;
  type T1347;
  type T1348;
  type T1349;
  type T1350;
  type T1351;
  type T1352;
  type T1353;
  type T1354;
  type T1355;
  type T1356;
  type T1357;
  type T1358;
  type T1359;
  type T1360;
  type T1361;
  type T1362;
  type T1363;
  type T1364;
  type T1365;
  type T1366;
  type T1367;
  type T1368;
  type T1369;
  type T1370;
  type T1371;
  type T1372;
  type T1373;
  type T1374;
  type T1375;
  type T1376;
  type T1377;
  type T1378;
  type T1379;
  type T1380;
  type T1381;
  type T1382;
  type T1383;
  type T1384;
  type T1385;
  type T1386;
  type T1387;
  type T1388;
  type T1389;
  type T1390;
  type T1391;
  type T1392;
  type T1393;
  type T1394;
  type T1395;
  type T1396;
  type T1397;
  type T1398;
  type T1399;
  type T1400;
  type T1401;
  type T1402;
  type T1403;
  type T14
```

S7. The parameter and result type profile of a formal subprogram parameter must be used to help resolve an overloaded actual parameter (RM 8.7/7-8 and RM 8.7/19):

```
function F return INTEGER is ... end F;      -- F1
function F return FLOAT is ... end F;       -- F2
```

```
generic
  type T is private;
  with function F return T;
package P is ... end P;
```

```
package P1 is new P (INTEGER, F);
package P2 is new P (FLOAT, F1);
```

Both instantiations are legal. The F in P1 resolves to F1; the F in P2 resolves to F2.

Changes from July 1982

S8. Parameter modes must be identical for formal and actual subprogram parameters.

S9. An enumeration literal matches a formal subprogram parameter.

Changes from July 1980

S10. The subtype of a subprogram's formal parameter or result type is not considered when attempting to find a match to a formal subprogram parameter.

S11. Entries are allowed as actual parameters.

Legality Rules

- L1. The actual parameter corresponding to a formal subprogram must be a subprogram or an entry having the parameter and result type profile of the formal parameter (i.e., the number of formal parameters must be the same, corresponding formal parameters must have the same base type, and if the formal subprogram parameter is a function, then the actual parameter must be a function or an enumeration literal; if the formal subprogram parameter is a procedure, the actual parameter must be a procedure or an entry.) In addition, corresponding parameters must have the same mode (RM 12.3.6/1).
- L2. If the formal parameter has a default subprogram specified with a box and there is no actual parameter, there must be exactly one (directly visible) subprogram, enumeration literal, or entry having the same designator as the formal subprogram and the same parameter and result type profile (RM 12.3.6/3). In addition, corresponding parameters must have the same mode (RM 12.3.6/1).

Test Objectives and Design Guidelines

- T1. Check that predefined operators may be passed as actual generic subprogram parameters.

Implementation Guideline: In particular, check that "/=" and the attributes 'SUCC, 'PRED, 'IMAGE, and 'VALUE can be passed. Include a case where the formal parameter is "=".

Check that a default subprogram name is used when there is no actual parameter (see also T7 for tests when the default is a box).

Check that an actual parameter cannot be omitted if no default subprogram is specified.

- T2. Check that entries may be passed as actual generic subprogram parameters, and the correct behavior is obtained when they are invoked.

Check that an expression in the name of the entry is only evaluated if there is no actual parameter.

Check that default names (and default expressions; see also 12.3.1/T23) are evaluated in the order given by the generic declaration.

- T3. Check that enumeration literals (both identifiers and character literals) may be passed as actual subprogram parameters.
- T4. Check that the formal and actual parameter and result type profiles must not be different, nor may the modes of corresponding parameters be different.

Implementation Guideline: Check with cases in which the formal and actual parameters disagree on just one of the following aspects:

- one is a procedure and one is a function;
- the base type of one parameter is different (check when the base type is either a formal generic type or a nongeneric type);
- the number of parameters is different, and the omitted parameters have defaults.

Implementation Guideline: Check that the modes are not considered when finding a match for the actual parameter.

- T5. Check that the following differences between formal and actual subprograms do not invalidate a match:

- different parameter names;
- different parameter constraints;
- one parameter constrained and the other unconstrained (for array, record, access, and private types);
- presence or absence of an explicit In mode indicator;
- different type marks used to specify the type of parameters.

- T6. Check that the default expressions of a formal subprogram's formal parameters are used when the formal subprogram is called in the instantiated unit (rather than any default associated with the actual subprogram's parameters).

Check that any constraints specified for the actual subprogram's parameters are used in place of any constraints associated with the formal subprogram's parameters.

Check that when the formal subprogram's parameters are specified with a formal generic type, the constraints (if any) of the corresponding actual subprogram's parameters are used.

- T7. Check that when a default subprogram is specified with a box, a subprogram directly visible at the point of the instantiation is used (rather than a subprogram visible at the point of the generic unit's declaration).

Check that the modes of corresponding parameters must match.

Check that the differences listed in T5 do not affect a match.

Implementation Guideline: The matching subprogram should sometimes be an operator symbol, and the matching operator symbol should sometimes be only visible via a use clause and sometimes only implicitly declared.

Check that more than one matching default subprogram is illegal.

- T8. Check that the parameter and result type profile of the formal parameter is used to resolve the actual parameter or the default (when the default is given either by name or as a box) (see IG 8.7.b/T59).

Implementation Guideline: The profile should sometimes depend on a formal type parameter declared earlier in the same generic formal part, and sometimes should depend on a formal parameter of an enclosing generic unit.

Chapter 13

Representation Clauses and Implementation-Dependent Features

13.1 Representation Clauses

Semantic Ramifications

S1. This section covers miscellaneous general issues concerning representation clauses. Later subsections cover:

- a. allowed alternate representations for a type,
- b. forcing occurrences,
- c. representations for derived types, and
- d. the pragma PACK.

S2. The extent to which representation clauses are accepted by an implementation must be specified in Appendix F (RM 13.1/10). An implementation is allowed to reject every representation clause (AI-00361), and consequently, every compilation unit that contains a representation clause. An implementation can even reject a representation clause that specifies what would normally be the default representation. For example:

```
type DT is new INTEGER;
for DT'SIZE use 16;
```

The representation clause for DT can be rejected even if INTEGER'SIZE is 16 for a particular implementation (AI-00361) and even though in the absence of the representation clause DT'SIZE would be 16.

S3. Of course, if an implementation supports representation clauses at all, a more likely reason for rejecting a clause is that it requires a representation that the implementation is not prepared to support. For example:

```
type T is
  record
    I : INTEGER;
  end record;
for T use
  record
    I at 0 range 3 .. 3 + INTEGER'SIZE - 1; -- (1)
  end record;
```

The clause at (1) could be rejected if an implementation does not allow integer components to extend over word boundaries.

S4. A type representation clause applies either to a type or to a first named subtype (RM 13.1/3). This means that a type representation clause can be given for a name declared by a type declaration because such a name either denotes a type or a first named subtype. But a type representation clause cannot be given for every declaration that declares a type. In particular, even though a task declaration always declares a type, the declared name does not always denote a type; the declared name only denotes a type if the task declaration contains the word "type" (RM 9.1/2). If the name declared by a task declaration denotes a single task instead of a task type, no type representation clause can be given for the name. Similarly, no

type representation clause can be given for the anonymous array type declared by an object declaration, e.g.:

```
S : array (1..5) of INTEGER;
```

No type representation clause can be given for S, since S denotes an object, not a type.

S5. No type representation clause can be given for a generic formal type because the declaration of a type and its representation clause must occur within the same declarative part or package specification. Since the declaration of a generic formal type occurs within a generic formal part, not within a declarative part or a package specification (RM 12.1/2), no representation clause can be given for it.

S6. A representation clause cannot be given for a type mark declared by a subtype declaration because such a type mark denotes a subtype (RM 3.3.1/1):

```
type NEW_INT is new INTEGER;
subtype S1 is NEW_INT;
subtype S2 is NEW_INT range 0..15;
for S1'SIZE use 16;           -- illegal
for S2'SIZE use 4;           -- illegal
```

The representation clause for S1 is illegal even if NEW_INT'SIZE would normally be 16, because S1 denotes an (unconstrained) subtype, not a type. The clause for S2 is similarly illegal because S2 denotes a subtype.

S7. There can legally be more than one representation clause for the same type or first named subtype if the clauses do not determine the same aspects of the representation (RM 13.6/1). For example:

```
type T1 is (E1, E2, E3);
for T1 use (1, 5, 10);      -- define enumeration representation for T1
for T1'SIZE use 4;          -- define size specification for T1

type T2 is
  record
    B1 : BOOLEAN;
    B2 : BOOLEAN;
  end record;
for T2 use                  -- define record representation for T2
  record
    B1 at 0 range 0..0;
    B2 at 0 range 1..1;
  end record;
for T2'SIZE use 2;          -- define size specification for T2
```

S8. At most two representation clauses can be given for a type since a size specification can be given for any type class, and the other kinds of representation clauses (namely, collection size specifications, task storage size specifications, small specifications, enumeration representation clauses, and record representation clauses) can be given only to mutually distinct type classes (namely, access types, task types, fixed point types, enumeration types, and record types).

S9. Representation clauses for first named subtypes can take into account the constraint imposed on the base type. For example, consider the following declarations of first named subtypes:

```

type T1 is range 0..15;
type T2 is digits 5 range -10.0 .. 10.0;

type T3 is delta 0.1 range -10.0 .. 10.0;
type T4 is array (1..5) of T1;
type T5 is new INTEGER range 0..15;
type T6 is new STRING(1..5);

```

Only the values of the subtype need to be represented in stored values since any attempt to declare objects having a larger subtype is either illegal or raises CONSTRAINT_ERROR:

```

X1 : T1'BASE;                      -- illegal
X2 : T1 range T1'BASE'FIRST .. T1'BASE'LAST; -- CONSTRAINT_ERROR
X3 : T4'BASE (1..10);              -- illegal
X4 : T4 (1..10);                   -- illegal

```

Hence, the following representation clauses can be accepted by an implementation:

```

for T1'SIZE use 4;
for T2'SIZE use FLOAT'SIZE;        -- assuming FLOAT'DIGITS > 4
for T3'SIZE use 1 + 4 + 4;         -- sign + integral + fractional
for T4'SIZE use 5 * T1'SIZE;
for T5'SIZE use 4;
for T6'SIZE use 5 * CHARACTER'SIZE;

```

S10. An implementation must reject a representation clause that is illegal or impossible for every implementation. (An error diagnostic message ought to indicate a user error rather than an implementation limitation.) For example:

```

type T1 is range 0..15;
for T1'SIZE use 4;
for T1'SIZE use 5;                  -- illegal: size already determined

type T2 is (E1, E2, E3);
for T2'SIZE use 1;                  -- illegal: size too small

type T3 is
  record
    B1 : BOOLEAN;
    B2 : BOOLEAN;
  end record;
for T3 use
  record
    B1 at 0 range 2..2;
    B2 at 0 range 4..4;
  end record;
for T3'SIZE use 2;                  -- illegal: size too small

```

In the last example, an implementation could instead reject the record representation clause as being inconsistent with the size clause.

S11. A representation clause is illegal for a private type prior to its full declaration and for a type that has a subcomponent of such an incompletely declared type (RM 7.4.1/4). A representation clause can only be given after a private type has been completely declared. Thus, for example:

```

package PKG is
  type PT (D : INTEGER := 0) is private;
  type PT2 is private;
  for PT'SIZE use ...;           -- illegal
  type RT is                     -- incompletely declared
    record
      C : PT;
      D : PT2;
    end record;
  for RT'SIZE use ...;           -- illegal
private
  for PT'SIZE use ...;           -- illegal
  for RT'SIZE use ...;           -- illegal

  type PT (D : INTEGER := 0) is record ... end record;
  for PT'SIZE use ...;           -- ok
  for RT'SIZE use ...;           -- illegal

  type PT2 is record ... end record;
  -- RT is now completely declared
  for RT'SIZE use ...;           -- ok
end PKG;
use PKG;

type DT3 is new PT;
for DT3 use record ... end record; -- illegal: private type, not
-- a record type
for DT3'SIZE use ...;           -- ok

package body PKG is
  type DT4 is new PT;
  for DT4 use record ... end record; -- ok
  for DT4'SIZE use ...;           -- ok
end PKG;

```

S12. The syntax does not allow a representation clause after the occurrence of a body in a declarative part (RM 3.9/2), nor is such a clause allowed in a generic formal part (RM 12.1/2).

Changes from July 1982

S13. There are no significant changes.

Changes from July 1980

S14. A type representation clause is no longer allowed for the anonymous type declared by a task declaration that declares a single task.

S15. A representation clause is allowed in the visible part of a package.

S16. A task storage size specification can no longer be given in a task specification.

Legality Rules

L1. A representation clause need not be accepted by any implementation (AI-00361).

L2. A type representation clause can only be given for a name declared explicitly in a task type declaration or in a type declaration (RM 13.1/3) other than a private type declaration (RM

- 7.4.1/4), incomplete type declaration (RM 3.8.1/4), or generic formal type declaration (RM 13.1/5).
- L3. A length clause specifying storage size can only be given for a task type or an access type that is not a derived type (RM 13.2/8-10).
 - L4. A length clause specifying SMALL can only be given for a fixed point type (RM 13.2/12).
 - L5. An enumeration representation clause is only allowed for an enumeration type (RM 13.1/3).
 - L6. A record representation clause is only allowed for a record type (RM 13.1/3).
 - L7. An address clause is only allowed for an object, subprogram, package, task unit, or entry. At most one address clause is allowed for any given object, subprogram, package, task unit, or entry (RM 13.1/4).
 - L8. A representation clause can only be given for an entity declared earlier and immediately within the same declarative part, package specification, or task specification (RM 13.1/5).
 - L9. A representation clause cannot be given after the occurrence of a body in a declarative part (RM 3.9/2).
 - L10. More than one representation clause can be given explicitly for a type only if the clauses specify different aspects of the representation. The legal combinations are a size specification combined with one of the following: a collection size specification, a task storage size specification, a small specification, an enumeration representation clause, or a record representation clause (RM 13.6/1).

Test Objectives and Design Guidelines

Since the set of representation clauses accepted by an implementation depends on the implementation, every test must be parameterized with respect to the representation clause applied to a particular type. For each form of representation clause accepted by an implementation, it will be possible to define a subtype indication and a corresponding representation clause value that can be used in more than one test. In particular, such repeated usage is possible when the test is checking that otherwise legal representation clauses are rejected when they are applied in illegal contexts (e.g., see test T1 or T6).

- T1. Check that a type representation clause cannot be given for the name declared by an object declaration (see IG 13.2.a/T1, IG 13.2.c/T1, IG 13.2.d/T1, IG 13.3/T1, and IG 13.4/T1) or by the declaration of a single task (see IG 13.2.a/T1 and IG 13.2.c/T1).

Check that a type representation clause cannot be given for a name declared by a subtype declaration (see IG 13.2.a/T1, IG 13.2.c/T1, IG 13.2.d/T1, IG 13.3/T1, and IG 13.4/T1).

Check that a type representation clause cannot be given for a type declared by a private type declaration or for a type having a component of an incompletely declared private type prior to the complete declaration for the type (see IG 13.2.a/T1, IG 13.2.c/T1, IG 13.2.d/T1, IG 13.3/T1, and IG 13.4/T1).

Check that a type representation clause cannot be given for a type declared by an incomplete type declaration prior to its full declaration (see IG 13.2.a/T1, IG 13.2.c/T1, IG 13.2.d/T1, IG 13.3/T1, and IG 13.4/T1).

Check that a type representation clause cannot be given for a generic formal type even if the clause appears in the visible part of a generic package specification (see IG 13.2.a/T1).

- T2. Check that an enumeration representation clause cannot be given for a type that is not an enumeration type (see IG 13.3/T1).

- T3. Check that a record representation clause cannot be given for a type that is not a record type (see IG 13.4/T1).
- T4. Check that two explicit representation clauses are not allowed for the same aspect of a type (see IG 13.6/T1).
- T5. Check that an address clause cannot be given for a generic subprogram, generic package, or exception (see IG 13.5/T4).
- T6. Check that a type representation clause cannot be given:
- in a package specification for a type declared in an inner package specification;
 - in a package or task specification, for a type declared in an enclosing package specification or declarative part;
 - in a package body for a type declared in the corresponding package specification;
 - after the occurrence of a body in a declarative part.

See IG 13.2.a/T3, IG 13.2.b/T3, IG 13.2.c/T3, IG 13.2.d/T3, IG 13.3/T3, and IG 13.4/T3.

- T7. Check that an address clause cannot be given (see IG 13.5/T6):
- in a package specification for an object, package, etc. declared in an inner package specification;
 - in a package or task specification, for an object, package, etc. declared in an enclosing package specification or declarative part;
 - in a package body for an object, package, etc. declared in the corresponding package specification;
 - after the occurrence of a body in a declarative part.
- T8. Check that a storage representation clause cannot be given for a name declared by a single task declaration (see IG 13.2.c/T1).
- T9. Check that a representation clause can be given in the visible or private part of a package for a type declared earlier in the same visible part.

13.1.a Multiple Representations for a Type

Semantic Ramifications

S1. An implementation can use more than one representation for values of a type whether or not representation clauses are given. For example:

```
X : INTEGER range 1 .. 10 := 10;
Y : INTEGER := X + 13;          -- no exception
```

The values of object X could be stored in a single byte, even though integer values in general will require more space. No exception is raised for Y's initialization expression because the addition operation is defined for the base type INTEGER, not for X's subtype. Even if a representation clause is given, such a clause only affects how values of the type are stored:

```
type T1 is range 0 .. 15;
for T1 use 4;
```


Expressions using values belonging to subtype T1 can compute values outside the range of T1 because these operations are declared for T1's base type. For example, the following is legal and cannot raise any exception if T1'BASE'LAST is at least 30:

```
Z : T1 := 2*T1'LAST/3;      -- final value is in T1's range
```

S2. Local variables having the same base type can have different physical representations because actual parameters can always be passed by copy, thereby allowing for any needed implicit representation conversions. In addition, the representation for a local variable need not be the same as the representation for an allocated variable since pointers can never designate local variables. (An implementation does not have to worry about pointers obtained by ADDRESS and UNCHECKED_CONVERSION. The use of such pointers is erroneous if any language or implementation rules are violated.)

S3. From the programmer's viewpoint, possible differences in representations of stored values have no effect on the behavior of a program except for values of representation attributes, e.g., SIZE; in the above example, if 8 bits are used to store values of X, then X'SIZE equals 8 even though INTEGER'SIZE might equal 32.

S4. An implementer may sometimes find it reasonable to have different representations for the same type in different contexts, to save on storage or time. Any such representation variations affect only the time and space behavior of a program, and the values of representation attributes such as SIZE.

S5. In many of the reasonable uses of multiple representations for a type, the representations will differ only:

- in the amount of storage allocated to objects or components that have that type. For example, if the component subtype for an array allows only values in the range 0..15, an implementation might well use only 4 bits to hold each component value. Similarly, each array object need only be allocated the space needed for its components, as determined by its index constraint.
- in the alignment of objects or components with respect to storage boundaries.
- in the use of unsigned or biased representations vs. signed representations (for integer types). (A biased representation could be used to represent the range 10..11 with a single bit, for example.)

Changes from July 1982

S6. There are no significant changes.

Changes from July 1980

S7. There are no significant changes.

Test Objectives and Design Guidelines

T1. Check whether an implementation gives different sizes to objects having the same base type.

Implementation Guideline: Check for scalar, array, and constrained record types.

13.1.b Forcing Occurrences

Semantic Ramifications

S1. Certain explicit occurrences of a type name or a value of a type are considered to

determine (force) the representation of a type. No representation clause is allowed after a forcing occurrence (RM 13.1/6-7). Specifically, certain occurrences of type mark TT are forcing occurrences for type T when TT is:

- a subtype of T (or T itself),
- a type that has a subcomponent of a subtype of T,
- an array type with index subtype T (AI-00321).

and when type TT is used in one of the following ways (RM 13.1/6):

- a. as the type mark in an object declaration;
- b. as the prefix of an attribute when the attribute is used in an expression or range;
- c. as an actual generic parameter;
- d. as the type mark in the declaration of a generic formal object or generic formal type;
- e. as the type mark in a qualified expression, explicit type conversion, membership test, or allocator, except when the expression occurs as the argument of a pragma (AI-00186 and AI-00322);
- f. when T is a subcomponent of type TT, the use of type TT in a representation clause for type TT (such a use is only a forcing occurrence for T; it is not a forcing occurrence for type TT);

In addition, the use of a value of type TT as an operand of a relational operator, membership test, or explicit type conversion (AI-00039) is a forcing occurrence for type T. (Use of a value of type TT is not otherwise a forcing occurrence for T; see examples below.)

S2. Occurrences of type TT or T are not forcing occurrences for type T in the following contexts (RM 13.1/6):

- a. in a subtype declaration, when TT is the type mark in the subtype indication, e.g.,

```
subtype T1 is T range 1..5;
subtype T2 is TT (1..5);
```

- b. in a derived type declaration, when TT is the parent type;
- c. in the declaration of an array type having component or index type TT;
- d. as the type mark in the declaration of a record component or discriminant;
- e. as the type mark in the declaration of the formal parameter of a subprogram or entry;
- f. in the subtype indication of a deferred constant declaration;
- g. when T is the simple name or attribute given in a representation clause;
- h. in the argument of a pragma (whether the pragma is recognized by the implementation or not; see AI-00186, AI-00322, and below);

S3. The declaration of an object is a forcing occurrence even when the object is declared in a generic unit:

```
type T is ...;
```

```

generic
package PKG is
    T1 : T;      -- (1) rep. of T now decided
end PKG;

```

for T use ...; -- (2) illegal (even if PKG is never instantiated)

S4. The explicit use of TT in an expression is normally a forcing occurrence:

```

type T is range 0..15;
type TT is array (1..5) of T;
I : INTEGER := TT'SIZE;      -- (1) size of T also decided
for T'SIZE use ...;         -- (2) illegal

```

The occurrence of TT in line (1) is a forcing occurrence for T, since T is a subcomponent of type TT. It would be a forcing occurrence even if the attribute were TT'LENGTH, whose value is not affected by the size of TT.

S5. Now suppose line (1) is replaced with a representation clause for TT:

```

for TT'SIZE use ...;      -- (1a)
for T'SIZE use ...;       -- (2)

```

The representation clause at (2) must be rejected. The occurrence of TT in line (1a) is not a forcing occurrence for type TT, but it is a forcing occurrence for type T, since T is a subcomponent of type TT.

S6. If the representation clauses for types T and TT appeared in the reverse order:

```

for T'SIZE use ...;      -- (3)
for TT'SIZE use ...;     -- (4)

```

then the clause at line (4) could be accepted because the occurrence of T in line (3) is not a forcing occurrence for type TT. For example, if T's size were specified as 8, then TT'SIZE could be specified as 40. (An even larger value could be specified for TT, since the size specification is just an upper bound.)

S7. The occurrence of a value of a type is not usually a forcing occurrence for the type:

```

type T is range 1..10;
type TT is array (1..2) of T;
function F (X : TT := (1,2)) return INTEGER; -- (1) uses a TT value
for T'SIZE use ...;                          -- (2) can be accepted

```

The use of TT to specify the type of F's formal parameter is not a forcing occurrence for TT or for T, nor is the occurrence of values of type T in TT's aggregate. The implicit conversion of the literals 1 and 2 to type T is not a forcing occurrence for type T because the name T does not occur explicitly.

S8. Sometimes the occurrence of a value is, however, considered to be a forcing occurrence for the value's type (AI-00039). For example, consider:

```

procedure PROC is
    type T is delta 1.0 range -10.0 .. 10.0;
    OBJ : constant BOOLEAN := PROC."="(1.1, 1.2);
    for T'SMALL use 10.0**(-BOOLEAN'POS(OBJ));

```

The initial value expression for OBJ is static since PROC."=" (the equality operator for type T) is an operator symbol that denotes a predefined operator and each actual parameter, 1.1 and

1.2, is a static expression. Hence, OBJ can be used in a static expression (RM 4.9/6). This means the expression in T's representation clause is allowed by RM 13.2/12. But AI-00039 says the representation clause is illegal because the occurrence of values of type T as operands of a relational operator is considered a forcing occurrence for type T. If this were not the case, an implementation would be placed in a difficult position because the representation clause determines the set of model numbers for type T, and, therefore, determines the accuracy of the relational expression. If T'SMALL is given the value 0.1, then OBJ must be FALSE, which means T'SMALL is specified to have the value 1.0 (a contradiction). If T'SMALL is given the value 1.0, then the initialization expression can be evaluated to have the value TRUE or FALSE, since 1.1 and 1.2 fall within the same model interval. But if OBJ is given the value TRUE, then the representation clause specifies that T'SMALL has the value 0.1 (again a contradiction). The only consistent set of values is for OBJ to have the value FALSE and T'SMALL to have the value 1.0. Instead of forcing this kind of analysis, AI-00039 simply says the representation clause is illegal because the initialization expression for OBJ is considered a forcing occurrence for T.

S9. AI-00039 says the occurrence of an operand of type T as the operand of an explicit conversion is a forcing occurrence for T:

```
OBJ2 : constant INTEGER := INTEGER (PROC."+"(1.1, 1.2));
```

Similarly, the occurrence of a value of type T as the operand in a membership test is a forcing occurrence:

```
OBJ3 : constant BOOLEAN := 1.1 in PROC."+"(1.1, 0.0) .. 1.2;
```

S10. A function returning the value of a type can appear in a default expression without being considered a forcing occurrence for the type:

```
type T is ...;
function F return T;
function G (X : T) return BOOLEAN;
type R is
  record
    Z : BOOLEAN := G(F);      -- not forcing occurrence for T
  end record;
for T use ...;                -- can be accepted
```

AI-00039 notes that the use of G in F's argument is not a forcing occurrence for type T, so the representation clause can be accepted. Of course, PROGRAM_ERROR will be raised if an object of type R is declared (without an explicit initial value) before the bodies of F and G have been elaborated.

S11. If the argument of a pragma is an expression and the expression contains what would normally be considered a forcing occurrence for a type, then the pragma is ignored; its presence does not affect the legality of later representation clauses (AI-00186 and AI-00322). For example, consider:

```
type YES is new INTEGER range 1..10;
pragma OPTIMIZE (YES(4) + 5);      -- ignored
pragma PRIORITY (YES' POS(1));    -- ignored
pragma DEBUG (YES' (1));          -- ignored
for YES' SIZE use 5;               -- acceptable
```

A pragma is supposed to have no effect if its argument is not acceptable (RM 2.8(9)). In the example with pragma OPTIMIZE, the argument is supposed to be an identifier, not an expression. Since the argument is not acceptable, the pragma is to have no effect. In

particular, the occurrence of type name YES in the pragma's argument does not affect the legality of the later representation clause. Similarly, although the argument of the pragma PRIORITY is a static *universal_integer* expression, AI-00186 requires that the pragma be ignored because it would otherwise contain a forcing occurrence for type YES. Finally, if a pragma is not recognized by an implementation, it is to have no effect. In particular, the pragma DEBUG is not language defined and may not be recognized by some implementations. If the pragma is not recognized by an implementation, it has no effect on the legality of the later representation clause, even though the type name seems to occur in an expression. If the pragma is recognized by an implementation, and if the implementation requires an argument having an integer type, AI-00186 nonetheless requires that this occurrence of the pragma be ignored because it contains what would otherwise be considered a forcing occurrence for type name YES. The pragma,

```
pragma DEBUG(YES);
```

would not be considered to contain a forcing occurrence because YES is not used in an expression.

S12. An expression in a representation clause for a type is not allowed to contain a forcing occurrence for the type (AI-00371). For example:

```
type T is delta 1.0 range -10.0 .. 10.0;
for T' SMALL use T'DELTA/2;                -- illegal
```

```
type E is (EA, EB);
for E use (EA => E'POS(EA)-1, EB => E'POS(EB)); -- illegal
```

Changes from July 1982

S13. An occurrence of a type name in the argument of a pragma (e.g., in the argument of pragma PACK) is not a forcing occurrence.

Changes from July 1980

S14. The notion of a forcing occurrence is introduced and defined explicitly.

Legality Rules

L1. A representation clause for a type is illegal after a forcing occurrence for the type (RM 13.1/5-6). A forcing occurrence for a type T is one of the following uses of type TT, where TT is either the type T, a subtype of T, a type that has a subcomponent of type TT, or an array type or subtype with an index subtype that is a subtype of T (AI-00321):

- a. as the type mark in an object declaration;
- b. as the prefix of an attribute when the attribute is used in an expression or range;
- c. as an actual generic parameter;
- d. as the type mark in the declaration of a generic formal object or generic formal type;
- e. as the type mark in a qualified expression, an explicit type conversion, a membership test, or an allocator, except when the expression occurs as the argument of a pragma (AI-00186 and AI-00322);
- f. when T is a subcomponent of type TT, the use of type TT in a representation clause for type TT (such a use is only a forcing occurrence for T; it is not a forcing occurrence for type TT);

In addition, the use of a value of type TT as an operand of a relational operator, a membership test, or an explicit type conversion (AI-00039) is a forcing occurrence for type T.

- L2. A representation clause for a type is illegal if an expression in the clause contains a forcing occurrence for the type (AI-00371).

Test Objectives and Design Guidelines

Each of the following objectives is to be performed for each form of representation clause and each form of type name TT whose use is a forcing occurrence for type T (see R1, above). When possible, use representation clauses that can be accepted by an implementation if it were not for the preceding forcing occurrence.

- T1. Check that a representation clause for type T cannot be given after type TT is used in an object declaration.

- T2. Check that a representation clause for type T cannot be given after type TT is used as the prefix of an attribute, and the attribute is used in an expression or range that is not the argument of a pragma.

Implementation Guideline: Include use of an attribute such as LENGTH, whose value is unaffected by the chosen representation.

Implementation Guideline: Include use as a default expression in a subprogram specification (generic and nongeneric) and record component declaration.

- T3. Check that a representation clause for type T cannot be given after type TT is used in a generic instantiation as an actual generic parameter.

- T4. Check that a representation clause for type T cannot be given after type TT is used in the declaration of a generic formal object or generic formal type.

- T5. Check that a representation clause for type T cannot be given after type TT is used in a qualified expression, an explicit type conversion, or an allocator, or when TT is used as the type mark in a membership test.

- T6. Check that a representation clause is illegal if an expression in the clause contains a forcing occurrence for the type whose representation is being specified.

Implementation Guideline: Include the use of a nonrepresentation attribute for the type, e.g., the use of 'DELTA in a clause specifying 'SMALL for a fixed point type.

- T7. Check that a representation clause for type T cannot be given after a value of type TT is used as the operand in a membership test or relational operator.

- T8. Check that occurrence of a type name in an expression of a pragma is not considered a forcing occurrence for the type.

Implementation Guideline: Include checks using pragma names not recognized by any implementation.

Implementation Guideline: Check for all forms of representation clause: a size specification, a task storage size specification, a collection size specification, a small specification, an enumeration representation clause, a record representation clause, and an address clause.

13.1.c Representation of Derived Types

Semantic Ramifications

- S1. A derived type inherits the representation of its parent type to the extent that the parent type's representation has been specified with a representation clause (AI-00138). Whether or not some aspect of a derived type's representation is inherited, its representation can, in many circumstances, be redefined (AI-00138 and AI-00099; see also IG 13.2.d/S). For example:

```

type T is ...;
for T'SIZE use ...;    -- (1)

type DT is new T;      -- (2) inherits the rep. clause (1) of T
...                    -- (3) no forcing occurrences for DT
for DT'SIZE use ...;   -- (4) overrides the inherited representation

```

S2. If the representation of some aspect of a parent type has not been determined or has been determined by default (e.g., because the parent type was declared in another declarative part or because there has been a forcing occurrence), the derived and parent types can, but need not, have the same representation. For example:

```

type T is ...;
type DT is new T;      -- (1) representation not yet determined
for T'SIZE use ...;    -- (2)

```

The representation clause at (2) applies only to T, i.e., DT is not required to have the same size as T.

S3. Only a size specification is legal for a type that is derived from an access type (RM 13.2/8). For example:

```

type A is access INTEGER;
type DT is new A;
for DT'SORAGE_SIZE use 512;  -- illegal
for A'SORAGE_SIZE use 1024;  -- ok; also determines DT'SORAGE_SIZE
for A'SIZE use 32;           -- ok if 32 bits is enough
for DT'SIZE use 24;          -- ok if 24 bits is enough

```

S4. A size specification is legal for a type derived from a private type. Only the size specification is meaningful for private types (since the type class is "private," and not "record," "enumeration," etc.). Of course, the size specification might not be acceptable to an implementation, depending on the number of bits required for the full declaration of the type and the implementation's restrictions on size specifications. But, in general, if the size specification would be allowed for the full declaration, then it should be allowed for the derived type.

S5. If a parent type has derivable subprograms, then only a representation clause specifying size, task storage size, or the value of small can be given for the derived type (RM 13.1/3; the size of a collection cannot be specified for a derived access type in any case; RM 13.2/8). The intent of this rule is to prevent users from giving different record or enumeration representation clauses for the parent and derived types, thereby incurring potentially expensive implicit representation conversions when invoking the derived subprograms.

S6. If an implicit representation clause is present for a derived type, any expressions in the clause are not reevaluated for the derived type (AI-00138). For example,

```

type type T is ... end T;
for T'SORAGE_SIZE use F(INTEGER'SIZE);  -- F is a function
type DT is new T;

```

The expression specifying the storage size for a task type need not be static. The same representation clause is implicitly present for type DT, but even so, the expression is not evaluated again; DT has the same storage size as T.

Changes from July 1982

S7. There are no significant changes for derived types.

Changes from July 1980

s8. There are no significant changes for derived types.

Legality Rules

- L1. Only a representation clause specifying a type's size, storage size, or value of small can be given for a derived type if (user-defined) subprograms have been derived (RM 13.1/3 and RM 13.2/8).

Test Objectives and Design Guidelines

- T1. Check that a collection size cannot be specified for a derived access type (see IG 13.2.b/T1).
- T2. Check that an enumeration representation clause or a record representation clause cannot be given for a derived type if the parent type has derivable subprograms (see IG 13.3/T11 and IG 13.4/T10).
- T3. Check that the representation of a parent type is inherited by a derived type if the parent type's representation was determined by a representation clause or pragma PACK.
- T4. Check that the representation of a derived type can be respecified even for an aspect that is inherited from the parent type.
Implementation Guideline: The representation clause for the parent type should occur before the derived type's declaration.
Implementation Guideline: Include a check that the parent and derived types can have different representations if representation clauses for both types occur after the derived type declaration.
Implementation Guideline: Check that a representation clause can be given for a derived type when the parent type is declared in another unit.
- T5. Check that a size specification is allowed for a derived private type (see IG 13.2.a/T21, /T31, /T41, /T51, /T61, /T71, /T81, and /T91).
- T6. Check that a storage size specification for a task type is not evaluated again for a derived task type.

13.1.d The pragma PACK

Semantic Ramifications

S1. The pragma PACK specifies that storage minimization should be the main criterion when selecting the representation of a record or an array type. The only representation clauses affected by the pragma PACK are the size specification and the record representation clause. The order of the representation clauses and the pragma PACK for a type is irrelevant prior to the forcing occurrence for that type. If the pragma appears after a forcing occurrence for the type, the implementation must ignore the pragma (RM 2.8/9). For example:

```

type T is
  record
    B1 : BOOLEAN;
    B2 : BOOLEAN;
    B3 : BOOLEAN;
  end record;
for T use
  record
    B1 at 0 range 2..2;
  end record
-- (1)

```



```

      B2 at 0 range 4..4:
end record:

```

```

...
A : T;
pragma PACK (T);

```

-- (2)
-- (3)
-- (4)

There is a forcing occurrence of T at (3). The pragma at (4) must be ignored since it appears after the forcing occurrence, i.e., an implementation could choose to place component B3 at position 5, making T'SIZE equal 5.

S2. Now suppose the pragma is moved to position (2). The effect of the pragma is implementation dependent, but presumably, if the clause at (1) has been accepted, the implementation can equally well place component B3 at positions 0, 1, or 3, so that the overall length of T is 4 bits.

S3. Now suppose the pragma is given prior to the representation clause. In this case, an implementation can obey the pragma and reject the subsequent record representation clause if it is inconsistent with the storage representation determined in accordance with the pragma. An implementation can also choose to obey both the pragma and the representation clause even though the pragma occurs first, i.e., the implementation could put components B1 and B2 in the positions specified by the representation clause and place component B3 in accordance with the pragma. This flexibility is consistent with the view in AI-00361 that an implementation is free to decide under what circumstances it will accept a representation clause.

S4. Another pragma whose position is governed by the rules for a representation clause is the CONTROLLED pragma. The only restriction is that the pragma is not allowed for a derived type (RM 4.8/11), i.e., is ignored if a derived type is named (see RM 2.8/9).

Changes from July 1982

S5. The pragma PACK now takes, as the argument, a simple name that is a type; an expanded name is not allowed.

S6. The pragma PACK is allowed to affect the mapping of components onto storage as well as the gaps between consecutive storage components.

Changes from July 1980

S7. There are no significant changes.

Test Objectives and Design Guidelines

T1. Check whether a pragma PACK is obeyed for certain kinds of types, e.g.,

- an array having a BOOLEAN or CHARACTER component type; check whether the minimum space is used;
- a record having components that are just of type BOOLEAN or type CHARACTER; check whether the minimum space is used;
- a record having a combination of BOOLEAN components and components having a small integer subtype;
- an array whose component subtype allows storage within a single byte.

T2. Check that a pragma PACK need not be obeyed when its argument is an expanded name.

T3. Check that a pragma PACK is ignored after a forcing occurrence.

T4. Check whether occurrence of a pragma PACK prior to a record representation clause makes the representation clause illegal.

T5. Check that a pragma PACK is not considered illegal if it occurs after a forcing occurrence.

13.2 Length Clauses

Semantic Ramifications

S1. Each form of length clause is discussed separately in subsequent subsections. This section covers material that is common to all length clauses.

S2. Only a *simple expression* is allowed in a length clause. This syntactic restriction has no consequences of practical interest, since the simple expression must have a numeric type, and any form of expression that has a numeric type and uses only *predefined* operators and function calls will always be a simple expression (see RM 4.4/2). However, if a relational operator or logical operator is overloaded to deliver a numeric type, then it is possible to write nonstatic numeric expressions that are not, syntactically, *simple_expressions*:

```
package PECULIAR is
  type T is new INTEGER;
  function "<" (L, R : T) return T;
end PECULIAR;

with PECULIAR; use PECULIAR;
procedure P is
  task type TT;
  for TT'SORAGE_SIZE use T'(5) < 3;    -- illegal
end P;
```

The representation clause is illegal because it is syntactically an expression, not a simple expression. The expression must be enclosed in parentheses to be a simple expression.

S3. The prefix of the attribute that appears in a length clause must be the simple name of a type. An expanded name, or the name T'BASE, is not allowed (AI-00300; not yet approved).

S4. An implementation-defined attribute is not allowed as the attribute designator in a length clause because RM 13.2/3 says only SIZE, STORAGE_SIZE, and SMALL are allowed. The intent is that implementation-defined representation specifications be supported by pragmas.

Changes from July 1982

S5. A length clause is not allowed for a task unit that denotes a single task.

S6. The wording has been clarified to limit which attributes can appear in length clauses. In particular, an implementation-defined attribute is not allowed.

Changes from July 1980

S7. The form of expression that appears after use in a length clause must be a simple expression instead of an expression.

Legality Rules

L1. The attribute designator in a length clause must be one of the following: SIZE, STORAGE_SIZE, or SMALL (RM 13.2/3).

L2. The prefix of the attribute in a length clause must denote a type or a first named subtype, but the type must not be a generic formal type, an incomplete type, or an incompletely declared private type (RM 13.2/3, RM 13.1/5, RM 7.4.1/4, and RM 3.8.1/4; see also IG 13.1/S).

- L3. The prefix of the attribute in a length clause cannot be an expanded name or the attribute T'BASE; it must be a simple name (AI-00300; not yet approved).
- L4. The expression in a length clause must have a numeric type (RM 13.2/3).

Test Objectives and Design Guidelines

- T1. Check that a length clause is not allowed for any attribute designator other than SIZE, STORAGE_SIZE, or SMALL.

Implementation Guideline: In particular, check the following attributes:

- for a prefix that denotes an object: ADDRESS;
- for a prefix that denotes an integer type: FIRST (to indicate the lowest value in a biased representation) and STORAGE_SIZE;
- for a prefix that denotes a fixed or floating point type: MACHINE_EMAX, MACHINE_EMIN, MACHINE_MANTISSA, MACHINE_OVERFLOW, MACHINE_RADIX, MACHINE_ROUNDS, SAFE_EMAX, SAFE_LARGE, or SAFE_SMALL.

- T2. Check that the form of expression in a length clause must be a simple expression (see IG 13.2.b/T14 and IG 13.2.c/T14).

13.2.a Size Specifications

Semantic Ramifications

S1. A size specification for a type T gives an upper bound (in bits) for the size of objects of type T. In the absence of a size specification, the upper bound is determined by the implementation. An implementation must not use more than the specified maximum, but it is permitted to use fewer bits as long as it uses enough to uniquely represent each value of type T (or subtype T when T is a first named subtype; see IG 13.1.a/S). The following examples illustrate some minimum sizes.

```
type T1 is range 0..15;  -- minimum size is 4 (unsigned rep.)
type T2 is range 7..10;  -- minimum size is 2 (biased rep.)
```

If an implementation only supports signed integer representations, then a minimal size specification would be rejected, i.e., an implementation need not support the logically minimal required size.

S2. An implementation may determine whether a size specification for an array type includes only the components of an array or whether it must also include any array descriptors (like vectors) (see also IG 13.4/S). For an unconstrained array type T, the specified size must be large enough to represent the largest possible array of type T (i.e., the array whose bounds are 'FIRST' to 'LAST' of the corresponding index subtypes). For example:

```
type T3 is array (1..8) of BOOLEAN;           -- min. size is 8
subtype EIGHT is INTEGER range 1..8;
type T4 is
  array (EIGHT range <>) of INTEGER range 0..3; -- min. size is 8*2
```

Since a specified size is only an upper bound, array objects of type T4 need only be allocated enough bits to represent their actual components:

```
A5 : T4 (3..6);                               -- min. size is 4*2
```

S3. When a derived type declaration for T is given in terms of a subtype of the parent type, only the subtype values need to be represented, e.g.,

```

type T5 is new INTEGER range 0..15;           -- min. size is 4
type T6 is new CHARACTER
  range CHARACTER'VAL(0) .. CHARACTER'VAL(7); -- min. size is 3

```

S4. A size specification is legal for a subtype if the subtype has a static constraint, even if the subtype itself is not static (RM 13.2/6). For example (SD means static range, dynamic name):

```

SS : constant INTEGER range 0..15 := 7; -- SS is static
SD : INTEGER range 0..255 := 7;         -- SD is not static
DD : constant INTEGER range 0..SD := 3; -- DD is not static

type T1 is new INTEGER range 0..SS;      -- static range 0..7,
                                         -- minimum size is 3
type T2 is new INTEGER range 0..SD;      -- nonstatic range,
                                         -- size specification illegal
type T3 is new INTEGER range 0..DD;      -- nonstatic range,
                                         -- size specification illegal

type DT1 is new T1 range 0..3;            -- static subtype and range
                                         -- minimum size is 2
type DT2 is new T2 range 0..3;            -- static constraint
                                         -- size specification legal

```

A size specification is legal for DT2 even though T2 is a nonstatic subtype and even though DT2 is, technically speaking, therefore a nonstatic subtype (RM 4.9/11).

S5. A size specification is illegal for a type if any of its constraints, subcomponent constraints, or index subtypes are nonstatic. For example:

```

generic
  type T is range <>;                      -- nonstatic type
package PKG is
  type DT is new T;                        -- nonstatic type
  type A1 is array (T range <>) of INTEGER; -- nonstatic index
                                         -- subtype
  type A2 is array (1..8) of T;            -- nonstatic
  type A3 is new A1 (1..8);                -- nonstatic
end PKG;

```

Size specifications are illegal for DT, A1, A2, and A3 even if PKG is never instantiated. The index constraint for A3 is nonstatic because the index subtype is not static. RM 4.9/11. A representation clause is allowed for formal type T in any case. see IG 13.2/5.

S6. A component subtype constraint that depends on a discriminant is not legal. A size specification cannot be given for record containing such constraints. For example:

```

subtype TEN is INTEGER range 1 .. 10
type REC (D : TEN) is
  record
    C : STRING (1 .. D)
  end record
type D_REC is new REC (3)
for REC SIZE use 10*CHARACTER'SIZE
for D_REC SIZE use 10*CHARACTER'SIZE

```

Component C does not have a static constraint.

AD-A189 647

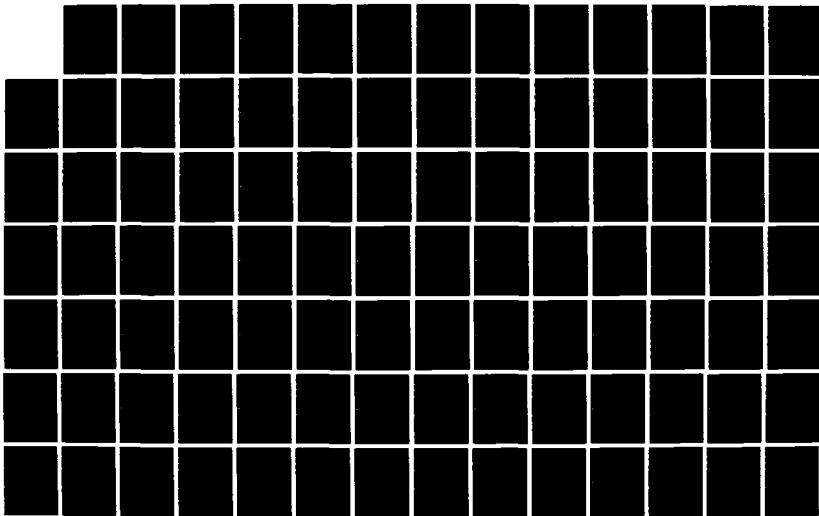
THE ADA (TRADE NAME) COMPILER VALIDATION CAPABILITY
IMPLEMENTERS' GUIDE VERSION 1(U) SOFTECH INC WALTHAM MA
J B GOODENOUGH DEC 86

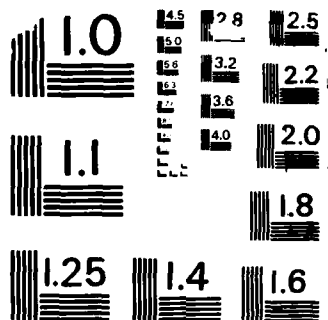
8/9

UNCLASSIFIED

F/G 12/5

NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

S7. A constraint specified for an access type applies to its designated type. If a size specification is allowed for an access type, it is allowed if a constraint is present, whether the constraint is static or not.

```
type ACC1_STR is access STRING (1..DD);           -- nonstatic constraint
type ACC2_STR is access STRING;
type D_ACC2_STR is new ACC2_STR (1..DD);
```

If a size specification is allowed for ACC2_STR, there is no reason it should not also be allowed for ACC1_STR and D_ACC2_STR.

S8. A size specification for a composite type T can affect the size of the storage area allocated to each component as well as the sizes of the gaps between the components. For example, when a component type has a static discrete subtype, each component need only occupy enough space to hold its range of possible values:

```
type T is array (1..5) of INTEGER range 0..15;
for T use 5*4;
```

Since each component need occupy only four bits (in unsigned representation), the size specification could be accepted for T. On the other hand, an implementation could reject the size specification on the basis that any subtype of INTEGER must occupy INTEGER'SIZE bits, so 20 bits are insufficient to hold the full array.

S9. If a component subtype is a composite type, a length clause for the containing type need not affect the representation of the component. For example:

```
subtype FIFTEEN is INTEGER range 0..15;
subtype FIVE    is INTEGER range 1..5;
type R is
  record
    B1 : FIFTEEN;           -- four bits suffice
    B2 : FIFTEEN;           -- four bits suffice
  end record;
type A is array (1..5) of R;
```

The two components of R could be packed into a single byte, but an implementation might not usually perform such packing because it takes additional time to access a component value. On the other hand, suppose the following is given as a representation clause for A:

```
for A use FIVE'LAST * (2 * FIFTEEN'SIZE);
X : R := A(3);
```

Suppose that FIFTEEN'SIZE equals 4, i.e., that an implementation is able to store values having subtype FIFTEEN in four bits. An implementation might consider the length clause for A to be an indication that the record type should be packed when used as a component of type A (since this is the only way the representation clause can be obeyed). But outside A, the implementation might use an unpacked representation by default. If so, the assignment required for X's initialization implies that a representation conversion is first performed for component A(3). (The expression in the representation clause could not be A'LENGTH * (2 * FIFTEEN'SIZE) since the prefix of LENGTH is not a scalar type, and so A'LENGTH is not allowed in a static expression; see RM 4.9/8).

S10. Of course, if the programmer gives a specific length clause for record type R, this clause should be obeyed when the type is used as a component of another type. For example, suppose the programmer indicates that byte alignment is desired for components of type R:

```

for R use
  record
    B1 at 0 range 0..7;
    B2 at 0 range 8..15;
  end record;

```

Then the minimum size for type R is 16, and a representation clause for type A should not be accepted if fewer than 16×5 bits is specified.

S11. A size specification is allowed for a task type. Such a specification refers to the representation used for a task object. Since the value of a task object designates a task (RM 9.2/2), such an object is likely to have the same representation as a pointer, and if so, restrictions on size specifications for task types are likely to be the same as restrictions on size specifications for access types. If a composite type has a component of a task type, then any size specification for the composite type will have to take into account the representation for task type values.

S12. A size specification is not allowed for a single task:

```
task SINGLE is ... end SINGLE;
```

SINGLE denotes a task, not a task type, and so cannot be named in a size specification.

S13. Since a size specification gives an upper bound on the number of bits used to represent values of the type, an implementation can use fewer bits if it so desires. If so, the value of the size attribute for the type may be less than the value given in the size specification (but cannot be more):

```

type T is ...;
for T use 32;
X : INTEGER := T'SIZE;

```

The value of X must be less than or equal to 32.

S14. When a sufficiently small collection size has been specified for an access type, access values might be represented in biased notation (i.e., as a relative value from a base address associated with the collection). If an implementation supports such a representation for access values, then a size specification for an access type can be used to obtain a biased pointer representation. (Such pointers are sometimes called offset pointers.) For example:

```

type T is access ...;
for T'SORAGE_SIZE use 32_768;  -- bytes
for T'SIZE use 16;

```

The size specification could be accepted for a machine such as the IBM S/370 for which ADDRESS'SIZE is 24.

S15. The RM imposes no requirement on the relative order in which a collection size and size specification should appear. In the previous example, the size specification might be rejected if it occurs before the storage size specification on the grounds that access values generally require 24 bits. On the other hand, an implementation could wait until the representation of T must be determined (e.g., by a forcing occurrence) before deciding whether the size specification can be accepted. If this strategy is used, then the collection size specification can appear after the size specification.

Changes from July 1982

S16. A size specification is explicitly allowed for a first named subtype.

S17. A size specification is allowed to affect the amount of storage allocated for each component of a composite type.

Changes from July 1980

S18. A size specification is not allowed for a composite type if a constraint on any subcomponent of the type is not static.

S19. A size specification is not allowed for an unconstrained array type if an index subtype is not static.

Legality Rules

The following restrictions apply to size specifications of the form:

for T'SIZE use simple_expression;

- L1. T must denote a type or a first named subtype, but the type must not be a generic formal type, an incomplete type, or an incompletely declared private type (RM 13.2/3, RM 13.1/5, RM 7.4.1/4, and RM 3.8.1/4; see also IG 13.1/S).
- L2. T cannot be an expanded name or the attribute T'BASE; it must be a simple name (AI-00300; not yet approved).
- L3. The expression must be static and must have an integer type (RM 13.2/5).
- L4. The value of the expression must be at least equal to the minimum number of bits needed to uniquely represent the values of (sub)type T (RM 13.2/5).
- L5. Any constraints on T and on its subcomponents must be static. If T is an unconstrained array type, its index subtypes must also be static (RM 13.2/6).
- L6. A size specification can be rejected if an implementation chooses not to support it (AI-00361).
- L7. Two size specifications cannot be given for the same type (RM 13.6/1).

Test Objectives and Design Guidelines

T1. Check that a size specification cannot be given for:

- a prefix that is an expanded name;
- the prefix T'BASE;
- a name declared by an object declaration;
Implementation Guideline: Include a check for an object declared with a constrained array type definition, since the base type is anonymous in this case.
- a name declared by a single task declaration;
- a name declared by a subtype declaration;
- a type declared by a private type declaration prior to the full declaration of the type;
- a type having a subcomponent of an incompletely declared private type, prior to the complete declaration of the composite type;
- a type declared by an incomplete type declaration prior to its full declaration;
- a type having a subcomponent of an incompletely declared private type, prior to the complete declaration of the composite type;

- a generic formal type.

Implementation Guideline: The size specification should not occur in the generic formal part.

Implementation Guideline: In each case, use a clause that, if possible, would be allowed for the actual type or for the completely declared type.

T2. Check that two size specifications cannot be given for the same type.

Implementation Guideline: Check for each kind of type: enumeration, integer, fixed point, floating point, array, record, access, derived private, and task type. When possible, the first size specification should be one that is accepted by the implementation; the second specification should be identical to the first.

T3. Check that a size specification cannot be given:

- in a package specification for a type declared in an inner package specification;
- in a package or task specification, for a type declared in an enclosing package specification or declarative part;
- in a package body for a type declared in the corresponding package specification;
- after the occurrence of a body in a declarative part.

T4. Check that a size specification cannot be given after a forcing occurrence for a type (see IG 13.1.b/T1-T7).

T5. Check that a size specification can be given after an occurrence of the type name in an expression of a pragma (see IG 13.1.b/T8).

T6. Check that the expression in a size specification must be static and must have an integer type.

T11. Check that a size specification cannot be given for a type that has a nonstatic constraint. In particular, check:

- a derived enumeration, integer, fixed point, or floating point type that has a nonstatic range constraint or a nonstatic parent type mark.

- an array type (derived or not) that has a nonstatic index constraint.

Implementation Guideline: Include a case where the index subtype is not static but the discrete range in the index constraint is static. Include a check that A'RANGE, A'FIRST, and A'LAST is not allowed when A is an array type or object, even if A has a static index constraint.

- a derived record or private type that has a nonstatic discriminant constraint.

Implementation Guideline: Include a case where the discriminant value is static but the discriminant subtype is not.

T12. Check that a size specification cannot be given for an unconstrained array type that has a nonstatic index subtype.

Implementation Guideline: Include a check for a derived unconstrained array type.

T13. Check that a size specification cannot be given for an array, record, or derived private type that has a subcomponent with a nonstatic constraint. In particular, check:

- an array type that has a nonstatic component or subcomponent constraint.
- a record type that has a component or subcomponent with a nonstatic constraint.
- a record type that has a component or subcomponent constraint that depends on a discriminant.

- a derived record or private type that has a subcomponent with a nonstatic constraint.
- a derived constrained or unconstrained array type that has a subcomponent with a nonstatic constraint.

Implementation Guideline: Check for the following forms of constraint and component type: scalar type (range constraint), array type (index constraint), record or private type (discriminant constraint). Note the various forms of nonstatic constraint checked in T11.

Size Specifications for Enumeration Types

T21. Check whether a size specification can be given for an enumeration type and that operations on values of such a type are not affected by the representation clause. In particular, if a size specification is allowed, check that:

- a similar but not necessarily identical specification can be given for a derived enumeration type.
- a similar specification can be given for a derived private type whose full declaration is an enumeration type.
- the size specification can be given in the visible or private part of a package for a type declared in the visible part.

Implementation Guideline: In checking that the clause does not affect the specified behavior of the type, check that there are no semantic anomalies when the type is used to declare an object, subprogram parameter, array component, or record component. Check that a value of the type can be used correctly in a relational expression or with the attributes PRED, SUCC, POS, VAL, IMAGE, and VALUE. Check that all these operations are performed correctly even when the type is passed as a generic actual parameter. Check that implicit representation conversions are performed correctly for derived subprograms.

Implementation Guideline: Repeat the checks for size specifications given in a generic unit.

T22. Repeat the checks of T21 for the smallest size appropriate for a signed and for an unsigned representation.

T23. If a size specification can be given for an enumeration type and an enumeration representation clause can also be given, repeat the checks of T21 when an enumeration representation clause has been given together with a size specification.

Implementation Guideline: The size specification should appear after the enumeration representation clause.

T24. Repeat the checks of T23 for the smallest size appropriate for a signed and for an unsigned representation.

T25. Check that a size specification is rejected if it is too small for an enumeration type when an enumeration representation clause is given for the type.

Check that a size specification is rejected if it is too small to represent each value of an enumeration type that has been given a user-defined representation.

Implementation Guideline: When an enumeration representation clause is present, the size specification should occur after the enumeration clause.

T26. If a size specification can be given after an enumeration representation clause, check whether it can also be given before the same clause.

Size Specifications for Integer Types

T31. Check whether a size specification can be given for an integer type and that operations on values of such a type are not affected by the representation clause. In particular, if a size specification is allowed, check that:

- a similar but not necessarily identical specification can be given for a derived integer type.
- a similar specification can be given for a derived private type whose full declaration is an integer type.
- the size specification can be given in the visible or private part of a package for a type declared in the visible part.

Implementation Guideline: In checking that the clause does not affect the specified behavior of the type, check that there are no semantic anomalies when the type is used to declare an object, subprogram parameter, array component, or record component. Check that a value of the type can be used correctly in a relational expression, arithmetic operation, or with the attributes PRED, SUCC, POS, VAL, IMAGE, and VALUE. Check that all these operations are performed correctly even when the type is passed as a generic actual parameter. Check that implicit representation conversions are performed correctly for derived subprograms.

Implementation Guideline: Repeat the checks for size specifications given in a generic unit.

- T32. Repeat the checks of T31 for the smallest size appropriate for a signed and for an unsigned representation.
- T35. Check that a size specification is rejected if it is too small for an integer type.

Size Specifications for Floating Point Types

- T41. Check whether a size specification can be given for a floating point type and that operations on values of such a type are not affected by the representation clause. In particular, if a size specification is allowed, check that:

- a similar but not necessarily identical specification can be given for a derived floating point type.
- a similar specification can be given for a derived private type whose full declaration is a floating point type.
- the size specification can be given in the visible or private part of a package for a type declared in the visible part.

Implementation Guideline: In checking that the clause does not affect the specified behavior of the type, check that there are no semantic anomalies when the type is used to declare an object, a subprogram parameter, an array component, or a record component. Check that a value of the type can be used correctly in a relational expression or an arithmetic expression. Check that all these operations are performed correctly even when the type is passed as a generic actual parameter. Check that implicit representation conversions are performed correctly for derived subprograms.

Implementation Guideline: Repeat the checks for size specifications given in a generic unit.

- T42. Repeat the checks of T41 for the smallest size appropriate for a signed and for an unsigned representation.
- T45. Check that a size specification is rejected if it is too small for a floating point type.

Size Specifications for Fixed Point Types

- T51. Check whether a size specification can be given for a fixed point type and that operations on values of such a type are not affected by the representation clause. In particular, if a size specification is allowed, check that:

- a similar but not necessarily identical specification can be given for a derived fixed point type.
- a similar specification can be given for a derived private type whose full declaration is a fixed point type.

- the size specification can be given in the visible or private part of a package for a type declared in the visible part.

Implementation Guideline: In checking that the clause does not affect the specified behavior of the type, check that there are no semantic anomalies when the type is used to declare an object, a subprogram parameter, an array component, or a record component. Check that a value of the type can be used correctly in a relational or arithmetic expression. Check that all these operations are performed correctly even when the type is passed as a generic actual parameter. Check that implicit representation conversions are performed correctly for derived subprograms.

Implementation Guideline: Repeat the checks for size specifications given in a generic unit.

- T52. Repeat the checks of T51 for the smallest size appropriate for a signed and for an unsigned representation.
- T53. If a size specification can be given for a fixed point type and a specification of small can also be given, repeat the checks of T51 when a small specification has been given together with a size specification.
- T54. Repeat the checks of T53 for the smallest size appropriate for a signed and for an unsigned representation.
- T55. Check that a size specification is rejected if it is too small for a fixed point type when no specification of SMALL has been given.
- Check that a size specification is rejected if it is too small to represent each value of a fixed point type that has been given a specification of SMALL.
- T56. If a size specification can be given after a small specification, check whether it can also be given before the same size specification.

Size Specifications for Array Types

- T61. Check whether a size specification can be given for an array type and that operations on values of such a type are not affected by the representation clause. In particular, if a size specification is allowed, check whether:

- the specified size can imply that gaps between components are compressed or eliminated.

Implementation Guideline: Check for arrays with a BOOLEAN component type and a STRING component (when the string does not occupy an integral number of words).

- the specified size can imply that array components are stored in a smaller amount of space than would be used for an object declaration having the component subtype.

Implementation Guideline: Declare a component with an integer, enumeration, array, and record subtype.

For each allowed size specification, check that:

- a similar but not necessarily identical specification can be given for a derived array type.
- a similar specification can be given for a derived private type whose full declaration is an array type.
- the size specification can be given in the visible or private part of a package for a type declared in the visible part.

Implementation Guideline: In checking that the clause does not affect the specified behavior of the type, check that there are no semantic anomalies when the type is used to declare an object, a subprogram parameter, an array component, or a record component. Check that a value of the type can be used correctly in a relational

or logical expression. Check that components can be set and read. Check that all these operations are performed correctly even when the type is passed as a generic actual parameter and/or a subprogram parameter.

- T62. Repeat the checks of T61 for size specifications given in a generic unit.
- T63. Repeat the checks of T61 for a type that has derivable subprograms, and check that any implicit representation conversions are performed correctly when the derived subprograms are called.
- T64. Repeat the checks of T61 when the component type itself has been given a size specification.
- T65. Repeat the checks of T62 when the component type itself has been given a size specification.
- T66. Repeat the checks of T63 when the component type itself has been given a size specification.
- T67. Check that a size specification is rejected if it is too small for an array type.

Size Specifications for Record Types

Interactions with record representation clauses are tested in IG 13.4/T61.

- T71. Check whether a size specification can be given for a record type and that operations on values of such a type are not affected by the representation clause. In particular, if a size specification is allowed, check whether:

- the specified size can imply that gaps between components are compressed or eliminated.

Implementation Guideline: Check for records with BOOLEAN components and STRING components (when the string does not occupy an integral number of words).

- the specified size can imply that array components are stored in a smaller amount of space than would be used for an object declaration having the component subtype.

Implementation Guideline: Declare components with integer, enumeration, array, and record subtypes.

For each allowed size specification, check that:

- a similar but not necessarily identical specification can be given for a derived record type.
- a similar specification can be given for a derived private type whose full declaration is a record type.
- the size specification can be given in the visible or private part of a package for a type declared in the visible part.

Implementation Guideline: In checking that the clause does not affect the specified behavior of the type, check that there are no semantic anomalies when the type is used to declare an object, a subprogram parameter, an array component, or a record component. Check that a value of the type can be used correctly in a relational expression. Check that components can be set and read. Check that all these operations are performed correctly even when the type is passed as a generic actual parameter and/or subprogram parameter.

- T72. Repeat the checks of T71 for size specifications given in a generic unit.
- T73. Repeat the checks of T71 for a type that has derivable subprograms, and check that any implicit representation conversions are performed correctly when the derived subprograms are called.

- T74. Repeat the checks of T71 when the component type itself has been given a size specification.
- T75. Repeat the checks of T72 when a component type itself has been given a size specification.
- T76. Repeat the checks of T73 when a component type itself has been given a size specification.
- T77. Check that a size specification is rejected if it is too small for a record type.

Size Specifications for Access Types

- T81. Check whether a size specification can be given for an access type and that operations on values of such a type are not affected by the representation clause. In particular, if a size specification is allowed, check that:

- a similar but not necessarily identical specification can be given for a derived access type.
- a similar specification can be given for a derived private type whose full declaration is an access type.
- the size specification can be given in the visible or private part of a package for a type declared in the visible part.

Implementation Guideline: In checking that the clause does not affect the specified behavior of the type, check that there are no semantic anomalies when the type is used to declare an object, a subprogram parameter, an array component, or a record component. Check that a value of the type can be used correctly in a relational expression. Check that designated objects can be accessed. Check that all these operations are performed correctly even when the type is passed as a generic actual parameter. Check that implicit representation conversions are performed correctly for derived subprograms.

Implementation Guideline: Repeat the checks for size specifications given in a generic unit.

- T83. If a size specification can be given for an access type and a collection size specification can also be given, repeat the checks of T81 when both a collection size and a size have been specified.

Implementation Guideline: The collection size specification should appear before the size specification.

- T84. Repeat the checks of T83 for the smallest size appropriate for a signed and for an unsigned representation.
- T85. Check that a size specification is rejected if it is too small for an access type (for the default representation).

Check that a size specification is rejected if it is too small to represent each value of an access type for which a collection size has been specified.

Implementation Guideline: When a collection size has been specified, the size specification should occur after the collection size specification.

- T86. If a size specification can be given after a collection size specification, check whether it can also be given before the same collection size specification.
- T87. If a size specification can be given for a derived, unconstrained access type, check that it can also be given for a derived access type whose parent type is constrained with a nonstatic index or discriminant constraint.

Size Specifications for Task Types

T91. Check whether a size specification can be given for a task type and that operations on values of such a type are not affected by the representation clause. In particular, if a size specification is allowed, check that:

- a similar but not necessarily identical specification can be given for a derived task type.
- a similar specification can be given for a derived private type whose full declaration is a task type.
- the size specification can be given in the private part of a package for a type declared in the visible part.

Implementation Guideline: In checking that the clause does not affect the specified behavior of the type, check that there are no semantic anomalies when the type is used to declare an object, a subprogram parameter, an array component, or a record component. Check that a value of the type can be used correctly in a relational expression or entry call. Check that all these operations are performed correctly even when the type is passed as a generic actual parameter. Check that implicit representation conversions are performed correctly for derived subprograms.

Implementation Guideline: Repeat the checks for size specifications given in a generic unit.

T95. Check that a size specification is rejected if it is too small for a task type.

13.2.L Collection Size Specifications

Semantic Ramifications

S1. A collection size specification is not permitted for a derived access type because the derived type shares its collection with its parent access type (see RM 3.4/9 and RM 13.2/8). For this reason, if a collection size is specified for a parent type, the derived type has the same collection size, even if the collection size for the parent type occurs after the derivation. For example:

```
type T is access STRING;
type DT is new T;
for T'SORAGE_SIZE use 32_768;
```

Since STORAGE_SIZE has been specified for parent type T, DT'SORAGE_SIZE = T'SORAGE_SIZE = 32_768.

S2. If the specified collection size is too small or too large for an implementation to support it, the implementation can either:

- reject the length clause,
- if too large, raise STORAGE_ERROR when the collection is allocated,
- if too small, raise STORAGE_ERROR when an allocator fails to find sufficient free space within the collection.

S3. If a collection size is specified, an implementation may be able to represent access values in fewer bits (as offset pointers). If so, a size specification may be given for an access type that specifies fewer bits than would usually be needed for an access value. Such a size specification could only be allowed if the collection size is specified with a static expression.

S4. Only a simple expression is allowed in a length clause. This syntactic restriction has no consequences of practical interest, since the simple expression must have a numeric type, and

any form of expression that has a numeric type and uses only *predefined* operations will always be a simple expression (see RM 4.4/2). However, if a relational operator or logical operator is overloaded to deliver a numeric type, then it is possible to write nonstatic expressions that are not, syntactically, *simple_expressions*:

```
package PECULIAR is
  type T is new INTEGER;
  function "<" (L, R : T) return T;
end PECULIAR;

with PECULIAR; use PECULIAR;
procedure P is
  type TT is access INTEGER;
  for TT'SORAGE_SIZE use T'(5) < 3;      -- illegal
end P;
```

The *representation clause* is illegal because it is syntactically an expression, not a simple expression. The expression must be enclosed in parentheses to be a simple expression.

Changes from July 1982

S5. There are no significant changes.

Changes from July 1980

S6. A collection size specification is not allowed for a type derived from an access type.

S7. The collection size specification also indicates the storage space needed to contain all objects designated by values of other types derived from the access type, directly or indirectly.

Legality Rules

The following restrictions apply to collection size specifications of the form:

for T'SORAGE_SIZE use simple_expression;

- L1. T must denote an access type, but not a derived access type (RM 13.2/8).
- L2. T cannot be an expanded name or the attribute T'BASE; it must be a simple name (AI-00300; not yet approved).
- L3. The expression must have an integer type (RM 13.2/8).
- L4. A collection size specification can be rejected if an implementation chooses not to support it (AI-00361).
- L5. Two collection size specifications cannot be given for the same access type (RM 13.6/1)

Test Objectives and Design Guidelines

T1. Check that a collection size specification cannot be given for:

- a prefix that is an expanded name;
- the prefix T'BASE;
- a derived access type or a derived private type whose full declaration is an access type;
- a name declared by an object declaration having an access type;
- a name declared by a subtype declaration;

- a type declared by a private type declaration prior to the full declaration of the type;
- a type declared by an incomplete type declaration prior to its full declaration;
- a generic formal access type.

Implementation Guideline: The size specification should not occur in the generic formal part.

Implementation Guideline: In each case, use a clause that, if possible, would be allowed for the actual type or for the completely declared type.

- T2. Check that two collection size specifications cannot be given for the same type.
- T3. Check that a collection size specification cannot be given:
- in a package specification for a type declared in an inner package specification;
 - in a package or task specification, for a type declared in an enclosing package specification or declarative part;
 - in a package body for a type declared in the corresponding package specification;
 - after the occurrence of a body in a declarative part.
- T4. Check that a collection size specification cannot be given after a forcing occurrence for type T (see IG 13.1.b/T1-T7).
- T5. Check that a collection size specification can be given after an occurrence of the type name in an expression of a pragma (see IG 13.1.b/T8).
- T11. Check whether a collection size specification can be given for an access type and that operations on values of the access type are not affected by the representation clause. In particular, if a size specification is allowed, check that a collection size specification can also be given in the visible or private part of a package for an access type declared in the visible part.
- Implementation Guideline:* In checking that the clause does not affect the specified behavior of the type, check that there are no semantic anomalies when the access type is used to declare an object or subprogram parameter. Check that a value of the type can be used correctly in a relational expression, and that an allocator can be evaluated correctly. Check that all these operations are performed correctly even for a derived access type and for an access type that is passed as a generic actual parameter. Check that allocators work correctly when called for a derived access type.
- Check that the expression in a collection size specification need not be static.
- Implementation Guideline:* Repeat the checks for a collection size specification given in a generic unit.
- T12. Check whether a size specification can be given together with a collection size specification (see IG 13.2.a/T83 and /T84).
- T13. If a size specification can be given after a collection size specification, check whether it can also be given before the same collection size specification (see IG 13.2.a/T86).
- T14. Check that the expression in a collection size specification must be a simple expression.
- T15. If a collection size is too small for holding a single value of the designated type, check whether the representation clause is rejected (when the expression has a static value) or if STORAGE_ERROR is raised when the representation clause is elaborated.
- If the collection size is large enough to hold some values of the designated type, check that STORAGE_ERROR is raised by an allocator when insufficient storage is available.
- T16. Check that if a collection size is given for a parent type, the derived type has the same collection size.

Implementation Guideline: Include a check when the parent type collection size is specified after the derived type declaration.

13.2.c Task Storage Size Specifications

Semantic Ramifications

S1. If a specified task storage size is larger or smaller than an implementation can support, the implementation can either:

- reject the length clause (e.g., if the value is static and negative, or if the specified value is static and greater than some implementation-defined maximum value).
- if too large, raise `STORAGE_ERROR` when the task's storage is allocated, or raise `NUMERIC_ERROR` (or `CONSTRAINT_ERROR`; see AI-00387) when the expression is evaluated.
- if too small, raise `STORAGE_ERROR` when a task activation/execution fails to find sufficient free space within the storage allocated for the task. (Of course, if `STORAGE_ERROR` is raised during the activation of a task, `TASKING_ERROR` will actually be propagated to the activating task; RM 9.3/7, /3).

S2. RM 13.2/14 shows an intent to allow considerable implementation freedom in deciding what storage is associated with a storage size specification. In particular, if a storage size specification is given for a task type and a dependent task has the same type as its master, `STORAGE_ERROR` could be raised when the dependent task is activated if the dependent task's storage is allocated from its master's storage. On the other hand, the storage associated with the dependent task could also be allocated independently, in which case, `STORAGE_ERROR` need not be raised.

S3. A task storage size can be specified for a derived task type since the name declared by a derived task type declaration denotes a type (AI-00422; not yet decided). A storage size specification is allowed for a derived task type even if a storage specification has been given for the parent type (AI-00361).

S4. A task storage size specification cannot be given for the name declared by a single task declaration, because such a name does not denote a task type (RM 9.1/2). For example:

```
task T is ... end T;
for T'STORAGE_SIZE use 1024;           -- illegal
```

The representation clause for T can't be given inside T's specification since T is not visible to (RM 8.3/5).

Changes from July 1982

S5. The prefix cannot be the name of a single task unit.

Changes from July 1980

S6. A task storage size specification can no longer be given in a task specification.

Legality Rules

The following restrictions apply to task storage size specifications of the form:

```
for T'STORAGE_SIZE use simple_expression;
```

- L1. T must denote a task type (RM 13.2/10).
- L2. T cannot be an expanded name or the attribute T'BASE; it must be a simple name (AI-00300; not yet approved).
- L3. The expression must have an integer type (RM 13.2/10).
- L4. A task storage size specification can be rejected if an implementation chooses not to support it (AI-00361).
- L5. Two task storage size specifications cannot be given for the same task type (RM 13.6/1).
- L6. A storage size specification for T cannot be given inside the specification for T (T is not yet visible; RM 8.3/5).

Test Objectives and Design Guidelines

T1. Check that a task storage size specification cannot be given for:

- a prefix that is an expanded name;
 - the prefix T'BASE;
 - a name declared by an object declaration having an access type;
 - a task declared by a single task declaration;
 - a name declared by a subtype declaration;
 - a type declared by a private type declaration prior to the full declaration of the type;
 - a type declared by an incomplete type declaration prior to its full declaration;
 - a generic formal type.
- Implementation Guideline:* The size specification should not occur in the generic formal part.
- inside a specification for the task type.

Implementation Guideline: In each case, use a clause that, if possible, would be allowed for the actual type or for the completely declared type.

T2. Check that two task storage size specifications cannot be given for the same type.

T3. Check that a task storage size specification cannot be given:

- in a package specification for a type declared in an inner package specification;
- in a package or task specification, for a type declared in an enclosing package specification or declarative part;
- in a package body for a type declared in the corresponding package specification;
- after the occurrence of a body in a declarative part.

T4. Check that a task storage size specification cannot be given after a forcing occurrence for type T (see IG 13.1.b/T1-T7).

T5. Check that a task storage size specification can be given after an occurrence of the type name in an expression of a pragma (see IG 13.1.b/T8).

T11. Check whether a task storage size specification can be given for a task type and that

operations on values of the task type are not affected by the representation clause. In particular, if a task storage size specification is allowed, check that a task storage size specification can also be given in the visible or private part of a package for a task type declared in the visible part.

Implementation Guideline: In checking that the clause does not affect the specified behavior of the type, check that there are no semantic anomalies when the access type is used to declare an object or subprogram parameter. Check that an entry call is accepted, and that storage within the task is allocated correctly. Check that all these operations are performed correctly even for a derived task type and for a task type that is passed as a generic actual parameter.

Check that the expression in a task storage size specification need not be static.

Check whether a different storage size can be specified for a derived task type.

T12. Check whether a size specification can be given together with a task storage size specification.

T14. Check that the expression in a task storage size specification must be a simple expression.

T15. If a specified task storage size is too small for any activation of the designated task, check whether the representation clause is rejected (when the expression has a static value) or if `STORAGE_ERROR` is raised when the task is activated and an attempt is made to allocate local storage for the task.

Check that `STORAGE_ERROR` is raised within the task when the specified amount of storage is found to be insufficient after the task has been successfully activated.

Check whether storage for a dependent task is included in the space associated with a task type's storage size specification.

Check whether storage for a collection is included in the space associated with a task type's storage size specification.

13.2.d Small Specifications

Semantic Ramifications

S1. Usually the expression in a small specification will have the type *universal_real*. In this case, the value of small is specified exactly. If the expression has a non-universal type, then its value might not be a model number, and even though the expression is a static expression, an implementation is allowed to choose any value within the model interval as the value of small. Of course, since the expression is a static expression, it is easy for the implementation to evaluate it exactly, but the RM does not, strictly speaking, require exact evaluation in this case.

S2. The specification of the range and `SMALL` of a fixed point type together determine the minimum number of bits needed to represent the model numbers of the type. A size specification and a small specification can be given for a fixed point type in either order. If the small specification occurs first, an implementation can choose a base type that has a particular size, e.g., an implementation might choose a 32-bit base type even though 16 bits would suffice. If a size specification occurs later that specifies 16 bits, the implementation could reject the size specification as being inconsistent with the selected base type. Similarly, if a size specification is given first (e.g., suppose 32 is specified), an implementation is still free to choose a base type that has a smaller number of bits. If it chooses too small a size, a later small clause (that requires a 32-bit representation) could be rejected.

S3. On the other hand, an implementation could defer the choice of a base type until there is a forcing occurrence for the type or until a representation must be chosen, e.g., at the end of the package specification or declarative part in which the type is declared. At that point, the

implementation could take into account any size and small specifications that have been given (including specifications for derived types; see below), and could determine if it is possible to choose a base type that satisfies all the requirements. The RM does not require such an approach. An implementation is free to choose a base type as soon as it finds it convenient to do so, and then reject later representation clauses that are inconsistent with this choice of base type. However, fixed point programmers are likely to want to specify both the size and small for a fixed point type, so implementations should choose strategies that allow both aspects of the type to be conveniently specified.

S4. A representation clause for a derived fixed point type is only allowed if the model numbers for the specified value of SMALL are representable values of the type; in addition, the value specified for SMALL cannot be greater than the delta of the type. For example:

```
type F1 is delta 1.0 range -15.0 .. 15.0;    -- F1'SMALL = 1.0
type DF1 is new F1 delta 4.0;
for DF1'SMALL use ...;
```

In deciding whether the representation clause is legal or not, we must take into account the following rules:

- RM 3.4/4 says the set of values of a derived type is a copy of the set of values for the parent type.
- RM 3.5.6/3 says an implementation of a real type must include the model numbers of the type and represent them exactly.
- HM 13.2/12 says the specification of *small* determines the value of *small* for the base type (see AI-00099; not yet decided).

Since the model numbers must be representable values of the type, and since the values of a derived type are determined by the parent type, no representation clause is allowed for a derived fixed point type unless the model numbers determined by the clause are representable values. With respect to the example, the representation clause is illegal if the model numbers for DF1 are not included in the set of model numbers for F1. What are the model numbers for F1?

S5. The model numbers for F1 are the model numbers of its base type (since F1 is a subtype that does not define a new set of model numbers; see RM 3.5.9/9 and RM 3.5.9/14; also IG 3.5.9/S). The model numbers for F1's base type include at least the values -15.0, -14.0, ..., 14.0, 15.0; these are exactly the model numbers for F1 if the base type has a mantissa length of 4. The set of values for the base type includes an additional negative number, -16.0, if F1 is represented in twos-complement notation.

S6. Now suppose that DF1'SMALL is specified to be 4.0. The set of values of type DF1'BASE is the same as the set of values for F1'BASE, but the model numbers for DF1 can have a reduced mantissa length of 2, so the smallest model number is $-3 * DF1'SMALL$, or -12, and the set of model numbers is -12.0, -8.0, -4.0, 0.0, 4.0, 8.0, and 12.0. Since this set of model numbers is contained in the set of values for DF1, such a value of DF1'SMALL is allowed.

S7. Now suppose that DF1'SMALL is specified to be 3.0. The mantissa for DF1 in this case must be at least 3; a mantissa value of 2 would imply that the largest model number is $2^{11} * DF1'SMALL$, i.e., 9.0, and the difference between 9.0 and the largest value belonging to subtype DF1 is $15.0 - 9.0 = 6.0$. Since 6.0 is greater than the specified value of SMALL, a mantissa length of 2 is insufficient to represent the set of model numbers required for DF1 (RM 3.5.9/6).

S8. A mantissa length of 3 implies that the largest model number is $2^{11} * DF1'SMALL =$

21.0, and so the set of model numbers is -21.0, -18.0, ..., 18.0, 21.0. If the chosen base type for the parent type includes these values, the representation clause is allowed. If, on the other hand, the parent type only has a mantissa length of 4, the parent type only includes values in the range -16.0 to 15.0. Therefore, 21.0 is not a representable value and the representation clause is illegal.

S9. If the representation clause for DF1'SMALL specifies the value 0.1, then the mantissa of DF1 must be 8 and the largest model number for DF1 will be $2\#1111_1111\# * 0.1 = 25.5$. (A mantissa length of 7 would not be allowed because then the largest model number would be $127 * 0.1$, which is insufficiently close to 15.0.) A specification of 0.1 will be allowed if the set of values for F1's base type includes the values -25.5, -25.4, ..., 25.4, 25.5. If these numbers are not represented exactly in F1's value set, the representation clause for DF1 must be rejected. Assuming that no representation clause is given for F1, F1's base type could be such that F1'BASE'SMALL is 1.0. In this case, the value 0.1, for example, is not in the set of values for F1, and hence, is not in the set of values for DF1, so DF1'SMALL could not be set to 0.1. On the other hand, F1's base type might be chosen so that F1'BASE'SMALL is 0.1, in which case, DF1'SMALL could be specified to be 0.1.

S10. Since choice of the base type affects the legality of a small specification for DF1, and since the representation clause for DF1 is not a forcing occurrence for the parent type, F1, the following sequence of declarations is allowed:

```
type F1 is delta 1.0 range -15.0 .. 15.0;    -- F1'SMALL = 1.0
type DF1 is new F1 delta 4.0;
for DF1'SMALL use 2.0/3.0;
for F1'SMALL use 3.0/9.0;
```

An implementation has several options at this point. If it is to accept both representation clauses, then F1'BASE'SMALL must be some value that allows both 2/3 and 4/9 to be represented exactly. The largest such value is 2/9, which is the least common multiple of 2/3 and 4/9. But there is no requirement for an implementation to choose such a value. The value chosen for the least significant bit of F1'BASE can be determined solely by the range given for F1 and the representation clause given for F1. If the implementation chooses to accept the clause for F1'SMALL, and also chooses F1's least significant bit to be equal to 4/9, then the representation clause for DF1 cannot be accepted. Since the clause for F1 occurs after the clause for DF1, it may prove awkward to reject the clause for DF1 after it has been processed. Since acceptance of representation clauses is implementation dependent (AI-00361), an implementation is free to decide the default representation of F1 when DF1 is declared or when DF1'SMALL is specified, and then, based on that choice, to reject the specification for F1.

S11. A more likely use of a derived fixed point type is in a context where the parent type's representation has already been determined (e.g., the parent type is declared in some library package). In this case, it is easy to decide whether a representation clause for the derived type can be accepted.

Changes from July 1982

S12. It is stated explicitly that the prefix of the attribute can denote a first named subtype.

Changes from July 1980

S13. There are no significant changes.

Legality Rules

The following restrictions apply to small specifications of the form:

```
for T'SMALL use simple_expression;
```

- L1. T must denote a fixed point type or first named subtype (RM 13.2/12).
- L2. T cannot be an expanded name or the attribute T'BASE; it must be a simple name (AI-00300; not yet approved).
- L3. The expression must be static and must have a real type (RM 13.2/12).
- L4. The value of the expression must not be greater than T'DELTA (RM 13.2/12).
- L5. A small specification can be rejected if an implementation chooses not to support it (AI-00361).
- L6. Two small specifications cannot be given for the same fixed point type (RM 13.6/1).

Test Objectives and Design Guidelines

T1. Check that a small specification cannot be given for:

- a prefix that is an expanded name;
- the prefix T'BASE;
- a name declared by an object declaration;
- a name declared by a subtype declaration;
- a type declared by a private type declaration prior to the full declaration of the type;
- a type declared by an incomplete type declaration prior to its full declaration;
- a generic formal fixed point type.

Implementation Guideline: The small specification should not occur in the generic formal part.

Implementation Guideline: In each case, use a clause that, if possible, would be allowed for the actual type or for the completely declared type.

T2. Check that two small specifications cannot be given for the same type.

T3. Check that a small specification cannot be given:

- in a package specification for a type declared in an inner package specification;
- in a package or task specification, for a type declared in an enclosing package specification or declarative part;
- in a package body for a type declared in the corresponding package specification;
- after the occurrence of a body in a declarative part.

T4. Check that a small specification cannot be given after a forcing occurrence for a type (see IG 13.1.b/T1-T7).

T5. Check that a small specification can be given after an occurrence of the type name in an expression of a pragma (see IG 13.1.b/T8).

T6. Check that the expression in a small specification must be static and must have a real type.

T7. Check that a small specification cannot be given for a derived fixed point type if the model numbers of the derived type are not included in the model numbers of the parent base type.

Implementation Guideline: Include checks where a small specification is given for the parent type, and the small specification for the parent type occurs both before and after the derived type declaration, and before and after the small specification for the derived type.

- T11. Check whether a small specification can be given for a fixed point type and that arithmetic operations for the type are performed correctly.

Check whether a small specification can be given for a derived fixed point type.

- T12. Check whether a size and small specification can both be given for a fixed point type (see IG 13.2.a/T53).

- T13. If a small specification can be given for a fixed point type, check that it can be given in the visible or private part of a package for a type declared in the visible part.

13.3 Enumeration Representation Clauses

Semantic Ramifications

- S1. The results produced by the POS and VAL attributes are not affected by an enumeration representation clause. For example:

```
type ENUM is (A, B, C);
for ENUM use (1, 2, 3);
```

With or without the representation clause, $\text{ENUM'POS}(A) = 0$ and $\text{ENUM'VAL}(1) = B$.

- S2. The aggregate used in an enumeration representation clause has no corresponding type declaration. No ambiguity exists even if an array type is visible that has a similar declaration. For example:

```
type ENUM is (A, B, C);
type ARR1 is array (ENUM) of INTEGER;
type ARR2 is array (ENUM) of INTEGER;
for ENUM use (1, 2, 3);                -- unambiguous aggregate
```

- S3. The requirement that the integer codes satisfy the predefined ordering relation for the type means that a code associated with a particular enumeration literal must be less than the code associated with another literal if and only if the first literal is less than the second, using the predefined "<" operation. For example, the representation clause for type ENUM could have been written as:

```
for ENUM use (C => 3, A => 1, B => 2)
```

- S4. The requirement that every choice in the representation aggregate is not redundant. For example, consider:

```
type ENUM is (A);
type ARR is array (ENUM) of INTEGER;
for ENUM use (ARR'RANGE => 1);        -- illegal
```

The choice, ARR'RANGE , is nonstatic because the prefix is not a scalar type (RM 4.9/8).

- S5. Within a generic unit, it is not possible to give a representation clause for a type derived from a generic formal discrete type since the actual type need not even be an enumeration type. It is possible to give an enumeration type representation clause for a derived type whose parent type occurs in a generic instantiation, although an implementation may decide not to allow such clauses:

```

generic
    type T is (<>);
package GP is
    type DT is new T;
end GP;

type ENUM is (A, B, C);

package INST is new GP (ENUM);

type D_ENUM is new INST.DT;           -- is an enumeration type;
for D_ENUM use (5, 10, 15);

```

The enumeration representation clause is allowed because INST.DT is an enumeration type with three enumeration literals (see AI-00398).

S6. An enumeration representation clause can only be given for a derived enumeration type whether or not a constraint applies to the parent type (AI-00422; not yet decided):

```

type ENUM is (A, B, C, D);
type D1 is new ENUM;
type D2 is new ENUM range A..C;
type D3 is new ENUM range A..D;

```

An enumeration representation clause can be given for types D2 and D3. Such a clause must give a representation for each value of the parent (type (i.e., for each value of D2'BASE and D3'BASE; see AI-00422; not yet decided).

Changes from July 1982

S7. All the choices given in the aggregate of an enumeration representation clause must be static.

Changes from July 1980

S8. A simple name of an enumeration type is required in the representation clause; an expanded name is no longer allowed.

S9. The integer codes specified for the enumeration type must satisfy the predefined ordering relation for the type (not necessarily a user-defined ordering relation).

Legality Rules

- L1. The type named in an enumeration representation clause must denote an enumeration type (RM 13.3/1 and RM 13.1/3).
- L2. The aggregate of an enumeration representation clause must be written as a one-dimensional array aggregate where the index subtype is the enumeration type and the component type is *universal_integer* (RM 13.3/3).
- L3. Integer codes must be specified for each enumeration literal of the type (RM 13.3/4).
- L4. Each enumeration literal must be given a distinct integer code (RM 13.3/4).
- L5. Each component value and choice given in the aggregate must be static (RM 13.3/4).
- L6. The *universal_integer* codes specified for the enumeration type must satisfy the predefined ordering relation of the type (RM 13.3/4), i.e., the code given for one literal must be less than the code given for another literal if and only if the first literal is less than the second using the predefined "<" operation for the type.

- L7. A forcing occurrence for an enumeration type is not allowed in the aggregate of a representation clause for the type (AI-00371).
- L8. An enumeration representation clause is not allowed for a derived enumeration type if the parent type has derivable subprograms (RM 13.1/3).

Test Objectives and Design Guidelines

- T1. Check that an enumeration representation clause cannot be given for:

- an expanded name that denotes an enumeration type;
- a name declared by an object declaration;
- a name declared by a subtype declaration;
- a type declared by a private type declaration prior to the full declaration of the type;
- an incomplete type prior to the full declaration of the type;
- a generic formal discrete type;
- a type that is not an enumeration type.

Implementation Guideline: In each case, use a clause that, if possible, would be allowed for the actual type or for the completely declared type.

- T2. Check that two enumeration representation clauses cannot be given for the same type.

Implementation Guideline: The two clauses should specify identical representations.

- T3. Check that an enumeration representation clause cannot be given:

- in a package specification for a type declared in an inner package specification;
- in a package or task specification, for a type declared in an enclosing package specification or declarative part;
- in a package body for a type declared in the corresponding package specification;
- after the occurrence of a body in a declarative part.

- T4. Check that an enumeration representation clause cannot be given after a forcing occurrence for the type (see IG 13.1.b/T1-T7).

- T5. Check that if an enumeration representation clause can be given, it can be given after an occurrence of the type name in an expression of a pragma (see IG 13.1.b/T8).

- T6. Check that the name of the enumeration type (or a subtype of the enumeration type) cannot appear as a choice in the aggregate or in one of the expressions.

Implementation Guideline: The name should be used in an attribute (e.g., VAL) that delivers a value of the required type.

- T11. Check that an enumeration representation clause cannot be given for a derived enumeration type if the derived type definition imposes a constraint or if the parent type has derivable subprograms.

Implementation Guideline: Write separate tests for these two cases.

- T12. Check that integer codes must be given for each enumeration literal of the type.

Implementation Guideline: Check that neither too many nor too few codes can be given.

Check that nonstatic integer codes are not allowed.

Implementation Guideline: Use nonstatic *universal_integer* expressions (see IG 4.10/S for ways of generating such expressions).

Check that a choice cannot be nonstatic.

- T13. Check that the same integer code cannot be given for two enumeration literals.

Check that the integer codes must obey the predefined ordering relation for the type.

Implementation Guideline: Include some aggregates in which choices do not appear in the order defined for the type, and for which an ordering operator has been explicitly declared.

Check that a choice in the aggregate must be a value of the enumeration type.

- T14. Check whether an enumeration representation clause can be given for an enumeration type. If so, check that such types can be used correctly in ordering relations, in indexing arrays, in attributes (see IG 3.5.5/T2), and in generic instantiations.

Implementation Guideline: Include cases where the integer codes have negative values and in which they do not have consecutive integer values.

Implementation Guideline: Combine this check with various forms of aggregate: all choices named (when some enumeration literals are character literals); some choices named; no choices named.

Check that an enumeration representation clause can be given in the visible or private part of a package for a type declared in the visible part.

Implementation Guideline: Repeat the checks for enumeration representation clauses given in a generic unit.

- T15. Repeat T14 for a derived enumeration type, including when the parent type has an enumeration representation clause given.

- T21. Check that the aggregate in an enumeration representation clause cannot be considered ambiguous.

Implementation Guideline: Declare more than one one-dimensional array type that has the enumeration type as its index subtype.

- T22. Check whether an enumeration representation clause can be given for a type derived from a type declared in a generic instantiation.

13.4 Record Representation Clauses

Semantic Ramifications

- S1. Although the alignment specified in an alignment clause must be a static expression, an address clause might not specify an address consistent with the required alignment:

```

type REC is
  record
    C : INTEGER;
  end record;
for REC use
  record
    at mod 2;           -- even address alignment
  end record;
OBJ : REC;
for OBJ use at 1;
```

The specified address is inconsistent with the specified alignment. In this case, since the address is given with a static expression, the inconsistency can be detected at compile time and the address clause can be rejected (since an implementation can place additional restrictions on

representation clauses; AI-00361). But the expression in an address clause need not be static. If it is not static, an implementation cannot detect the inconsistency until run time. An implementation could, of course, require that addresses be specified with static expressions (in which case there is no problem with rejecting the address clause at compile time), or an implementation could specify that any such inconsistency makes execution of the program erroneous, and PROGRAM_ERROR will be raised (AI-00337; not yet decided).

S2. The simple expression after at mod must have an integer type and means that the address modulo the expression must be zero. If SYSTEM.ADDRESS is not an integer type, then an implementation can refuse to support mod clauses. Equally well, it can give a definition that is implementation dependent.

S3. The example given in RM 13.4/9 for the record type PROGRAM_STATUS_WORD illustrates several subtle points. First of all, the record type has components that are arrays of BOOLEAN, e.g., SYSTEM_MASK. However, the record representation clause provides space only for the BOOLEAN components of these arrays, and not for any associated array descriptors (dope vectors). Actually, given only the record representation clause, an implementation could place the array descriptors after the specified components; it is the subsequent size specification for the record type that eliminates any space for the array descriptors. Hence, if an implementation accepts both the record representation clause and the size specification, then it may have to provide separate array descriptors when these array components are passed as actual parameters to subprogram formal array parameters. For example:

```

type A is array (NATURAL range <>) of BOOLEAN;

type R is
  record
    A1 : A (1..8);           -- static constraints for A1
    A2 : A (4..11);
  end record;

for R use                               -- assume STORAGE_UNIT = 8
  record
    A1 at 0 range 0..7;      -- OK since constraints of A1 and
    A2 at 1 range 0..7;      --   A2 are static
  end record;

for R'SIZE use 16;

R1 : R;

procedure P (X1 : in out A; X2 : in out A);
...

P (R1.A1, R1.A2);

```

Since P is declared to have unconstrained formal array parameters, the actual bounds of R1.A1 and R1.A2 must be passed (presumably in associated array descriptors) to the formal parameters X1 and X2 in the call to P. An implementation could either allow the array descriptors to be separate from the array components (which usually works), or it could construct copies of the actual parameters (but with array descriptors and components combined) and could pass the copies instead. On the other hand, the length clause could be rejected because there is no space for descriptors.

S4. A second point illustrated by the PROGRAM_STATUS_WORD example is that the size specified for a record component need only be large enough to hold the values possible for that component's subtype. Thus, the PROTECTION_KEY component is declared as INTEGER range 0..3 and is placed in a field of size 2. This size is less than INTEGER'SIZE, but is adequate to represent the values 0 through 3. An implementation is also permitted to use biased representations for record components, just as with size specifications for types (see IG 13.2.a/S). For example:

```

type R is
  record
    I : INTEGER range 7..10;
  end record;

for R use
  record
    I at 0 range 0..1; -- biased representation, where
                      -- logical 7 <=> physical 0 etc.
  end record;

```

If an implementation accepts such a record representation clause, it must supply the appropriate implicit representation conversion when such a component is accessed, assigned, or passed as an actual parameter. Of course, scalar types are always passed by copy (RM 6.2/6) and any other type can always be passed by copy (RM 6.2/7).

S5. A third point illustrated by the PROGRAM_STATUS_WORD example is that the alignment of a record component of type T need not be the same as for a separate variable of type T. For example, the INTEGER component CC is bit-aligned, whereas many implementations would always align separate INTEGER variables on halfword or word boundaries (depending on INTEGER'SIZE). Again, if an implementation accepts such a record representation clause, it must supply the appropriate implicit representation conversion when such a component is accessed, assigned, or passed as an actual parameter. The component can be passed by copy (RM 6.2/7).

S6. A component clause is illegal for a record component if any of the constraints on the component or on its subcomponents are nonstatic. However, as discussed previously, a component clause can specify a smaller size than base_type'SIZE for a statically constrained component provided that the size is sufficient to uniquely represent each value of the component's subtype and that the implementation supports the required representation. If not, the program must be rejected. For example:

```

SS : constant INTEGER range 0..15 := 7; -- SS is static
SD : INTEGER range 0..255 := 7;         -- SD is not static
DD : constant INTEGER range 0..SD := 3; -- DD is not static

type R is
  record
    I1 : INTEGER range 0..SS; -- static subtype
                                -- minimum size is 3
    I2 : INTEGER range 0..SD; -- nonstatic range,
                                -- component clause illegal
    I3 : INTEGER range 0..DD; -- nonstatic range,
                                -- component clause illegal
  end record;

```

```

generic
    type T is range <>;                -- nonstatic subtype

package PKG is
    L : constant T := T'FIRST;          -- nonstatic value
    type DT is new T;                   -- nonstatic subtype
    type ATI is array (T range <>) of INTEGER; -- nonstatic index
                                           -- subtype
    type AIT is array (1..8) of T;       -- nonstatic component type

    type RT is
        record
            T1 : T;                      -- nonstatic subtype
            DT1 : DT;                    -- nonstatic subtype
            ATI1 : ATI (1..8);           -- nonstatic index subtype
            AIT1 : AIT;                  -- nonstatic component
            T2 : T range 0..3;           -- nonstatic T
        end record;
    for RT use
        record
            -- component clauses for T1, T2, DT1, ATI1, and AIT1
            -- are illegal even if PKG is never instantiated
        end record;
end PKG;

```

Similar requirements exist for size specifications (see IG 13.2.a/S).

S7. An implementation can provide components in a record in addition to those explicitly declared in the record type definition. An extra component can be used, for example, to give the offset or bounds of another component, or to contain information that would otherwise be computed repeatedly at run time. An implementation is permitted to place restrictions on what record representation clauses it will support when such extra components are present. An implementation is permitted to create names for these extra components. These created names can be used only within the record representation clause. A created name must have the syntactic form of a name, e.g., a simple name or a machine-dependent attribute. Each name must be unique within the set of created and explicitly declared names for a particular record type, i.e., the implementation must create the names in a way that avoids conflicts. For example:

```

N1 : constant := ...;
N2 : constant := ...;

type R is
    record
        S1 : STRING (1 .. N1);
        S2 : STRING (1 .. N2);
    end record;

-- assuming that:  SYSTEM.STORAGE_UNIT = 8
--                INTEGER'SIZE = 16
--                CHARACTER'SIZE = 8

for R use
    record

```

```

S1'LENGTH at 0 range 0..15;
S1'OFFSET at 2 range 0..15;
S2'LENGTH at 4 range 0..15;
S2'OFFSET at 6 range 0..15;
S1 at 8 range 0 .. N1*CHARACTER'SIZE-1;
S2 at 8 + N1*CHARACTER'SIZE/SYSTEM.STORAGE_UNIT
    range 0 .. N2*CHARACTER'SIZE-1;
end record;

```

S8. A record type representation clause can be given for a record type, or for a type derived from a record type (as long as there are no derivable subprograms for the parent type; RM 13.1/3) (see AI-00422; not yet decided). If a constraint is given for the parent type, the representation clause nonetheless applies to the parent type, and the representation clause can (but need not) mention components that do not exist in the derived subtype: For example:

```

type REC (D : POSITIVE) is
  record
    B : BOOLEAN;
    case D is
      when 1..10 =>
        C1 : INTEGER range 0..7;
      when others =>
        C2 : STRING(1..50);
    end case;
  end record;

```

```

type D_REC is new REC(1);

```

A record representation clause can be given for D_REC, and the clause can even give a position for component C2 (AI-00422; not yet decided).

S9. The range in a component clause cannot be a range attribute because such an attribute is not static (its prefix cannot be a scalar type). Therefore, the only form of range allowed in a component clause has the form L..R.

Changes from July 1982

S10. The bounds of a range in a record representation clause need not have the same integer type.

S11. At most one component clause is allowed for each component of a record type.

Changes from July 1980

S12. The type name in a record representation clause must be a simple name; an expanded name or TBASE is no longer allowed.

Legality Rules

- L1. The type mark in a record representation clause must denote a record type (RM 13.1/3). (It cannot denote a derived record type if the parent subtype is constrained.)
- L2. The expression in an alignment clause must be static and have an integer type (RM 13.4/3).
- L3. The expressions in a component clause must be static. Each expression must have an integer type, but not necessarily the same integer type (RM 13.4/3).
- L4. The range in a component clause cannot have the form of a range attribute (RM 13.4/3).

- L5. An implementation may place restrictions on allowable values of the expression in an alignment clause (RM 13.4/4).
- L6. At most one component clause is allowed for each component of the record type (RM 13.4/6).
- L7. A component clause is only allowed for a component if every constraint on this component or on any of its subcomponents is static (RM 13.4/7 and AI-00132).
- L8. Each component clause must allow for enough storage space to represent every value of the component's subtype (RM 13.4/7).
- L9. Storage places within a record must not overlap except that components belonging to different variants may occupy the same space (RM 13.4/7).
- L10. A record representation clause is not allowed for a derived record type if the parent type has derivable subprograms (RM 13.1/3).
- L11. A record representation clause must be rejected if it violates any implementation-defined restrictions with respect to:
 - the allowable alignment of entire records (as specified in an alignment clause),
 - components overlapping storage boundaries,
 - the alignment of a component within the record,
 - the specified size for a component being less than the implementation's default size for that component, and the implementation not supporting the smaller size (e.g., when a biased representation would be required).

Test Objectives and Design Guidelines

- T1. Check that a record representation clause cannot be given for:
 - an expanded name that denotes a record type;
 - a name declared by an object declaration;
 - a name declared by a subtype declaration;
 - a type declared by a private type declaration prior to the full declaration of the type;
 - an incomplete type prior to the full declaration of the type;
 - a type having a subcomponent of an incompletely declared private type, prior to the complete declaration of the composite type;
 - a type that is not an enumeration type.

Implementation Guideline: In each case, use a clause that, if possible, would be allowed for the actual type or for the completely declared type.

- T2. Check that two record representation clauses cannot be given for the same type.

Implementation Guideline: The two clauses should specify identical representations.

- T3. Check that a record representation clause cannot be given:

- in a package specification for a type declared in an inner package specification;

- in a package or task specification, for a type declared in an enclosing package specification or a declarative part;
 - in a package body for a type declared in the corresponding package specification;
 - after the occurrence of a body in a declarative part.
- T4. Check that a record representation clause cannot be given after a forcing occurrence for the type (see IG 13.1.b/T1-T7).
- T5. Check that a record representation clause can be given after an occurrence of the type name in an expression of a pragma (see IG 13.1.b/T8).
- T6. Check that an expression in an alignment clause or a component clause must be static and must have an integer type.
- T7. Check that a component clause cannot be given more than once for a particular component, or for a nonexistent component.
- T8. Check that a component clause is not allowed for a component that has a nonstatic constraint or a subcomponent with a nonstatic constraint. In particular, check the following forms of constraint:
- a range constraint in which one of the expressions is not static.
Implementation Guideline: Check for ranges having enumeration, integer, floating point, and fixed point types.
 - a static range constraint imposed on a nonstatic scalar subtype.
 - an index constraint for a component having an array type when:
 - the index constraint contains a nonstatic expression.
Implementation Guideline: Include a case where the index constraint uses a discriminant of the enclosing record.
 - the index constraint only has static expressions, but an index subtype is nonstatic.
 - the index constraint is static but a subcomponent of the array has a nonstatic constraint.
 - a discriminant constraint for a component having a record or private type when:
 - the discriminant constraint contains a nonstatic expression.
Implementation Guideline: Include a case where the constraint uses a discriminant of the enclosing record type.
 - the discriminant constraint only has static expressions, but a discriminant subtype is nonstatic.
 - the discriminant constraint is static but a subcomponent has a nonstatic constraint.
 - the component is unconstrained but there is a nonstatic discriminant subtype.
- T9. Check that different components cannot be allocated overlapping storage space if they do not belong to different variants.

T10. Check that a record representation clause cannot be given for a derived record type with a constraint or whose parent type has derivable subprograms.

Check that a record representation clause cannot be given for the discriminants of a derived private type.

T11. Check that a range attribute cannot be used in a component clause.

T21. For a record component having an enumeration subtype, check whether a component clause can be given. If so, check whether the clause can specify less than the usual amount of space allocated to a variable having that subtype.

T22. For a record component having an integer subtype, check whether a component clause can be given. If so, check whether the clause can specify less than the usual amount of space allocated to a variable having that subtype.

Implementation Guideline: Check for signed, unsigned, and biased representations.

T23. For a record component having a floating point subtype with a digits constraint, check whether a component clause can be given. If so, check whether the clause can specify less than the usual amount of space allocated to a variable having that subtype.

T24. For a record component having a fixed point subtype with a specified delta and/or a range constraint, check whether a component clause can be given. If so, check whether the clause can specify less than the usual amount of space allocated to a variable having that subtype.

Implementation Guideline: Check for signed, unsigned, and biased representations.

T25. For a record component having an array subtype, check whether a component clause can be given. If so, check whether the clause can specify less than the usual amount of space allocated to a variable having that subtype.

Implementation Guideline: Check whether gaps between components can be suppressed, as for size specifications.

T26. For a record component having a record subtype, check whether a component clause can be given. If so, check whether the clause can specify less than the usual amount of space allocated to a variable having that subtype.

Implementation Guideline: Check whether gaps between components can be suppressed, as for size specifications.

T31. For a variant record type, check that components belonging to different variants can be given overlapping storage.

T41. Check whether an alignment clause can be given for a record representation clause.

T51. Check whether a record representation clause can be given for a derived record type.

T61. Check whether a size specification can be given for a type that also has a record representation clause.

13.5 Address Clauses

Semantic Ramifications

S1. A with clause naming the package SYSTEM must apply (in the sense of RM 10.1.1/4) to the unit containing an address clause. This does not mean that the with clause must be given for the compilation unit containing the address clause:

```

with SYSTEM;
package P is
    ...
end P;

package body P is
    X : INTEGER;
    for X use at ...;    -- legal

```

The address clause is allowed because the with clause given for the package specification applies to the body (RM 10.1.1/4). Similarly, the with clause applies to any subunit of P's body.

S2. Only a simple expression is allowed in an address clause. If a relational operator or a logical operator is overloaded to deliver a value of type SYSTEM.ADDRESS, then it is possible to write expressions that are not, syntactically, simple_expressions:

```

with SYSTEM;
package PECULIAR is
    function "<" (L, R : INTEGER) return SYSTEM.ADDRESS;
end PECULIAR;

with PECULIAR; use PECULIAR;
procedure P is
    OBJ : INTEGER;
    for OBJ use at 5 < 3;    -- illegal
    ...
end P;

```

The address clause is illegal because it is syntactically an *expression*, not a simple expression. The expression must be enclosed in parentheses to be a simple expression.

S3. The expression in an address clause need not be a static expression, in part because the type SYSTEM.ADDRESS might be a record or a private type. Moreover, an implementation is allowed to impose special requirements on the nature of allowed addresses (AI-00361). For example, an implementation might support address clauses only if the expression in the clause is static.

S4. An implementation may require that objects having certain types be aligned on certain address boundaries. For example, it might only be possible to allocate an object of type INTEGER to an even address. If so, an address clause that specifies a non-even address can be rejected. Similarly, a record representation clause might impose certain alignment requirements:

```

for REC use
    record at mod 8;
    end record;
for REC use at 65;    -- can be rejected

```

The inconsistency between the record and the address clause implies that one of them must be rejected.

S5. If the expression in an address clause is not static, an implementation cannot necessarily decide at compile time whether the specified address will be acceptable. In this case, an implementation has several options: it is free to reject the address clause on the basis that it cannot guarantee the alignment requirements will be satisfied, or it can check the alignment restrictions at run time and raise PROGRAM_ERROR if they are not obeyed (AI-00228 and AI-00337; not yet decided).

S6. An address clause for an object is not allowed after an occurrence of the name of the object (RM 13.1/8), although an occurrence in a pragma is not considered forcing (AI-00423; not yet decided):

```
TIME : INTEGER := 3;
X    : INTEGER := 3;
pragma OPTIMIZE (TIME);
for TIME use at ...;
pragma PRIORITY (X + 3);
for X use at ...;
```

The address clauses are allowed because TIME in the pragma OPTIMIZE does not denote the variable TIME and because the pragma PRIORITY must be ignored (its argument is not static). But use of the names TIME or X in a default expression, renaming declaration, etc., would make a subsequent address clause illegal.

S7. Since the ADDRESS attribute is a representation attribute (see RM 13.7.2), an occurrence of the attribute ADDRESS for a subprogram, a package, a task unit, or an entry makes a subsequent address clause for the entity illegal (RM 13.1/8).

S8. An address clause is not allowed after a subprogram body, even if the body acts as the declaration of the subprogram:

```
begin
  procedure P is ... end P;
  for P use at ...;           -- illegal
end;
```

The address clause is illegal because RM 3.9/2 does not allow any representation clause after a body.

S9. An address clause cannot be given for a library unit because a representation clause is not allowed, syntactically, between or after compilation units (RM 10.1/2), nor can the clause specify the address of a unit that contains the clause (RM 13.5/7).

S10. An address clause can be given inside a generic unit. If the generic unit is instantiated more than once and both instantiations are elaborated, the execution of the program may be erroneous (RM 13.5/8). The program is not necessarily erroneous:

```
generic
package GP is
  X : INTEGER;
  for X use at 15;
end GP;

with GP;
procedure PR is
begin
  declare
    package NP1 is new GP;
  begin
    ...
  end;

  declare
    package NP2 is new GP;
  begin
```

```

    end;
end PR;

```

Since the objects NP1.X and NP2.X exist at different times, the address clause does not have the effect of overlaying one object on the other. That is, even though the objects share the same storage, an assignment to one object never changes the value of the other object.

Address Clauses for Objects

S11. An address clause for an object is only allowed for an object declared by an object declaration or a single task declaration; it is not allowed for objects declared by other forms of declaration. To see this, consider the various forms for declaring objects (RM 3.2/2-7):

- an object declared by an object declaration or by a single task declaration,
- a formal parameter of a subprogram, an entry, or a generic subprogram,
- a generic formal object,
- a loop parameter,
- an object designated by a value of an access type, and
- a component or a slice of another object.

An address clause cannot be given for a formal parameter of a subprogram, an entry, or a generic subprogram because the formal parameter name is not declared, syntactically, within a declarative part (RM 13.5/7). Similarly, an address clause cannot be given for a generic formal object or a loop parameter because the object's declaration does not occur within a declarative part or a package specification. An address clause cannot be given for an object designated by an access type or for a component or a slice of another object because only a simple name can be given in an address clause. (A renaming declaration could be used to get around this limitation, but a name declared by a renaming declaration is not allowed in an address clause (RM 13.5/7).) Since we have now covered all the forms for declaring objects, it is clear that an address clause for an object can only be given if the object has been declared by an object declaration or a single task declaration. If the declared object is a constant and the address clause is accepted, this implies that the value of the constant is stored at the specified location.

S12. As noted above, an address clause can be given for an object declared by a single task declaration:

```

task T is ... end T;
for T use at ...;      -- ok

```

The clause specifies the address of the task body's code (RM 13.5/7), i.e., the code designated by task object T (see RM 9.1/2 and RM 9.2/2).

S13. A named number is declared by "a special form of object declaration," (RM 3.2/8) and so is arguably an object (AI-00263; not yet decided). If so, an address clause can be given for a named number. (This might be useful to force materialization of literals in locations that will be known at run-time.)

S14. An address clause for an object is not allowed after an occurrence of a name that denotes the object (RM 13.1/8). For example, occurrence of the object's name in a default expression, a renaming declaration, etc., would make a subsequent address clause illegal.

```

X : INTEGER;
Y : INTEGER renames X;      -- occurrence of X
for X use at ...;          -- illegal

```

Address Clauses for Subprograms

S15. An address clause can be given for a function. An enumeration literal is a function (RM 3.5.1/3) and so are certain attributes such as POS and VAL (RM 3.5.5/5-13). Since an enumeration literal can be declared in a declarative part or a package specification, an address clause is allowed for such a literal. But, of course, an implementation might choose to reject such a clause (AI-00361). An address clause cannot be given for one of the function attributes since a simple name is required.

S16. An address clause is allowed for an overloaded subprogram as long as only one of the visible subprograms is declared earlier in the same package specification or declarative part (RM 13.5/7). For example:

```

package P is
  procedure PR;
end P;

with P; use P;
package Q is
  procedure PR (X : INTEGER);
  for PR use at ...;      -- ok
end Q;

```

The address clause is allowed even though PR is overloaded.

S17. Similarly, an address clause is allowed if the subprogram is overloaded by an implicit declaration of a derived subprogram:

```

package P is
  type T is range 1..10;
  function F return T;
end P;

with P;
package Q is
  type NT is new P.T;
  function F (X : INTEGER) return NT;
  for F use at ...;      -- can be accepted
end Q;

```

S18. An address clause is also allowed for a subprogram declared in a task body even if it overloads an entry:

```

task type T is
  entry E;
end T;

task body T is
  function E return INTEGER;
  for E use at ...;      -- can be accepted

```

The key point is that the subprogram named in the address clause must have been declared explicitly by exactly one declaration occurring earlier in the same package specification or declarative part (RM 13.5/7).

S19. An address clause cannot be given for a generic subprogram because a generic unit is not a subprogram unit.

Address Clauses for Packages

S20. An address clause for a package does not specify the address of the package specification; the clause only gives the location of the body (RM 13.5/5).

Address Clauses for Task Units

S21. When an address clause is given for a single task, the clause specifies the address of the code associated with the task body (see RM 13.5/7), not the address of the task object (which designates the task and its associated code; RM 9.1/2 and RM 9.2/2). Note the difference here between giving an address clause for a single task and for a task object:

```
task T1 is
    ...
end T1;
for T1 use at ...;

task type T2 is
    ...
end T2;

XT2 : T2;
for XT2 use at ...;
```

The address clause for T1 specifies the location of the machine code for T1's task body. The address clause for XT2 specifies the location of the object that designates a task of type T2.

Changes from July 1982

S22. A compilation unit can contain an address clause only if a with clause naming the predefined package SYSTEM applies to the compilation unit. (Previously, a with clause had to be given directly for the compilation unit.)

S23. The address clause is only legal if exactly one declaration with the simple_name occurs earlier, immediately within the same declarative part, package specification, or task specification. (More than one such name can be visible.)

S24. A name declared by a renaming declaration is not allowed as the simple_name of an address clause.

Changes from July 1980

S25. The expression in an address clause need not be static, e.g., it can be an aggregate.

S26. The simple_expression has a value of type ADDRESS defined in the package SYSTEM instead of some integer type interpreted as an address.

Legality Rules

L1. The expression in an address clause must have the type SYSTEM.ADDRESS (RM 13.5/3).

- L2. A with clause naming the predefined package SYSTEM must apply to the compilation unit in which an address clause occurs (RM 13.5/3).
- L3. The name given in an address clause must denote an object (including a named number; see AI-00263 -- not yet decided), a subprogram, a package, a task unit, or a single entry (RM 13.5/4-6).
- L4. The name given in an address clause must be declared explicitly by exactly one subprogram declaration, generic instantiation, object declaration, number declaration, formal parameter declaration (of a subprogram, generic subprogram, or entry), generic formal object declaration, loop parameter specification, package specification, task specification, or single entry declaration (RM 13.5/7).
- L5. The name given in an address clause cannot be the name of a library unit (RM 13.5/7).
- L6. The name given in an address clause is only allowed if exactly one explicit declaration with this identifier occurs earlier, immediately within the same declarative part, package specification, or task specification (RM 13.5/7).
- L7. A name declared by a renaming declaration is not allowed as the simple name in an address clause (RM 13.5/7).
- L8. An address clause is not allowed for an object after an occurrence of a name that denotes the object, unless the occurrence is in a pragma (RM 13.1/8 and AI-00423; not yet decided).
- L9. An address clause is not allowed for a subprograms, a package, a task, or an entry after occurrence of an ADDRESS attribute for such an entity (RM 13.1/8).
- L10. An address clause is not allowed for a single task or a task object after an occurrence of a STORAGE_SIZE attribute for the task type or single task (RM 13.1/8).

Test Objectives and Design Guidelines

- T1. Check that the expression in an address clause must have the type SYSTEM.ADDRESS.
- T2. Check that an address clause is illegal if a with clause naming the predefined package SYSTEM does not apply to the unit containing the address clause.
Implementation Guideline: Check for objects, subprograms, packages, tasks, and entries.
- T3. Check that if an address clause is allowed, a with clause naming SYSTEM need not be given for the compilation unit containing the address clause as long as such a clause applies to the unit.
Implementation Guideline: Check for address clauses in package bodies, subprogram bodies, and subunits. Include a check for generic unit bodies and subunits.
- T4. Check that an address clause cannot be given for a named number, an exception, a formal parameter of a subprogram, entry, or generic unit, a generic formal object, a generic subprogram, a generic package, a loop parameter, an object designated by an access value, a slice, or a component of an object.

Check that an address clause cannot be given for a library unit or a generic unit.
- T5. Check that an address clause cannot be given for an expanded name or for a name declared by a renaming declaration.
Implementation Guideline: Include renamings of objects, subprograms, packages, tasks, and entries.
- T6. Check that an address clause cannot be given:
 - in a package specification for an object, a package, etc., declared in an inner package specification;

- in a package or task specification, for an object, a package, etc., declared in an enclosing package specification or a declarative part;
- in a package body for an object, package, etc. declared in the corresponding package specification;
- after the occurrence of a body in a declarative part.

Implementation Guideline: In particular, check for a subprogram body that also acts as the declaration of the subprogram.

- T7. Check that an address clause cannot be given for a subprogram if more than one subprogram with the same name is declared explicitly in the same package specification or declarative part.

Implementation Guideline: Include a generic instantiation and a renaming declaration as well as a subprogram declaration.

Check that if an address clause can be given for a subprogram, it can be given when the subprogram is overloaded by:

- a subprogram declared in an outer declarative region or library package,
- an entry declaration (when the subprogram is declared in the task body),
- an implicitly declared derived subprogram.

- T8. Check that the expression in an address clause must be a simple expression.

- T11. Check whether an address clause can be given for an object declared in a declarative part.

Implementation Guideline: Use a variable and constant having the following types: enumeration, integer, floating point, fixed point, array, record, access, private, limited private, and task.

Implementation Guideline: Check for declarative parts of subprograms, blocks, and package bodies.

- T12. Repeat T11 for generic units.

- T13. Check whether an address clause can be given for an object declared in a package specification.

Implementation Guideline: Use a variable and a constant having the following types: enumeration, integer, floating point, fixed point, array, record, access, private, limited private, and task.

Implementation Guideline: Include a check that the address clause can be given in the private part for an object declared in the visible part.

- T14. Repeat T13 for generic packages.

- T21. Check whether an address clause can be given for a subprogram declared in a declarative part by a subprogram declaration or a generic instantiation.

Implementation Guideline: Check for a declarative part of a block, a package body, a subprogram, and a task body.

- T22. Check whether an address clause can be given for a subprogram declared in a package by a subprogram declaration or a generic instantiation.

Implementation Guideline: Include a check that the clause can be given in the private part for a subprogram declared in the visible part.

- T31. Check whether an address clause can be given for a package declared in a declarative part by a package declaration or a generic instantiation.

Implementation Guideline: Check for a declarative part of a block, package body, subprogram, and task body.

- T32. Check whether an address clause can be given for a package declared in a package by a package declaration or generic instantiation.

Implementation Guideline: Include a check that the clause can be given in the private part for a package declared in the visible part.

T41. Check whether an address clause can be given for a task type or a single task declared in a declarative part.

Implementation Guideline: Check for a declarative part of a block, a package body, a subprogram, and a task body.

T42. Check whether an address clause can be given for a task type or a single task declared in a package.

Implementation Guideline: Include a check that the clause can be given in the private part for a task or a task type declared in the visible part.

13.5.1 Interrupts

Semantic Ramifications

S1. Although the RM requires that any entry parameters have mode In, no requirement is imposed on the types of these parameters. If the types seem to be inappropriate, an implementation can refuse to accept the address clause (since acceptance of representation clauses is implementation dependent; AI-00361).

S2. No special priority is associated with an entry just because it is an interrupt entry; the priority of the rendezvous is defined by the calling task (RM 9.8/5). When the calling task is the hardware task, the rendezvous is executed with the hardware task's priority. When the entry is called from a program unit, the rendezvous is performed with the priority of the calling task.

S3. An address clause can be given for an entry of a task type:

```
task type T is
  entry E;
  for entry E use 45;
end T;
```

The effect if more than one object of type T is declared is implementation dependent (see AI-00379).

S4. An address clause exists for an entry of a derived task type if it exists for the parent task type (AI-00292), e.g.:

```
type DT is new T;
type DDT is new DT;
```

If an address clause was given for an entry of task type T, it also exists for types DT and DDT. If objects of type T, DT, and DDT are declared, the effect of the address clauses is implementation-dependent (AI-00379).

S5. If an implementation supports address clauses for task entries, the occurrence of an interrupt means that the currently executing task will be preempted (if the currently executing task is not already being executed in response to a higher priority interrupt) (see AI-00032).

Changes from July 1982

S6. The priority of an interrupt entry is defined to be higher than the priority of the main program (as well as higher than the priority of any user-defined task).

Changes from July 1980

S7. If a select statement contains both an accept alternative for an interrupt entry and a terminate alternative, then an implementation may impose further requirements for the selection of the terminate alternative in addition to those given for normal select statements.

Legality Rules

- L1. If an address clause is given for an entry, the entry must only have parameters of mode in (RM 13.5.1/1).
- L2. An address clause cannot be given for an entry family (RM 13.5/6).

Test Objectives and Design Guidelines

- T1. Check that an address clause cannot be specified for an entry that has a parameter of mode out or mode in out.
- T2. Check that an address clause cannot be specified for an entry family.
- T3. Check that the name in an address clause for an entry cannot be an expanded name.
- T4. Check that an address clause for an entry cannot be given within the declarative part of the task body.
- T11. Check that if an implementation supports address clauses for entries, such a clause can be given for an entry of a task type as well as for an entry of a single task.

13.6 Change of Representation**Semantic Ramifications**

S1. An explicit representation clause can be given for a derived type even if a clause has been given for the same aspect of the parent type (AI-00371), but a record representation clause or an enumeration representation clause cannot be given if the parent type has derivable subprograms (RM 13.1/3).

Changes from July 1982

S2. There are no significant changes.

Changes from July 1980

S3. At most one representation clause is allowed for each aspect of a type.

Legality Rules

- L1. More than one representation clause can be given explicitly for a type only if the clauses specify different aspects of the representation. A size specification can be given for a specific type together with at most one of the following: a collection size specification, a task storage size specification, a small specification, an enumeration representation clause, or a record representation clause (RM 13.6/1).
- L2. A record representation clause or an enumeration representation clause cannot be given for a derived type if the parent type has derivable subprograms (RM 13.1/3) or if the parent subtype is constrained.

Test Objectives and Design Guidelines

- T1. Check that two explicit type representation clauses are not allowed for the same aspect of a type (see IG 13.2.a/T2, IG 13.2.b/T2, IG 13.2.c/T2, IG 13.2.d/T2, IG 13.3/T2, IG 13.4/T2).
- T2. Check whether a derived type can be given a representation that is different from the representation of its parent type (see IG 13.2.a/T21, /T31, /T41, /T51, /T61, /T71, /T81, and /T91, IG 13.2.c/T11, IG 13.2.d/T11, IG 13.3/T15, IG 13.4/T51).

13.7 The Package System

Semantic Ramifications

S1. Although a program library can contain at a given time only one version of package SYSTEM (i.e., only one value of SYSTEM_NAME, STORAGE_UNIT, MEMORY_SIZE, etc.), an implementation can provide various means of determining which version is to be used when compiling. The pragmas SYSTEM_NAME, STORAGE_UNIT, and MEMORY_SIZE provide one way of specifying different versions of package SYSTEM, but an implementation might have special commands that select from a set of possible SYSTEM packages.

S2. The meaning of the pragma MEMORY_SIZE is implementation dependent, e.g., whether the value refers to the total amount of memory available in the hardware, or to the amount of memory reserved for a particular program, is determined by the implementation.

S3. There is no requirement that a with clause for package SYSTEM apply to a unit that uses the pragma PRIORITY, even though the pragma has no effect if its value does not lie within the range defined by SYSTEM.PRIORITY (RM 2.8/8 and RM 9.8/3; see also AI-00197). Similarly, no with clause is needed for SYSTEM when writing a record representation clause even though a record component clause refers to storage units.

S4. The subtype PRIORITY need not be static; only the argument to pragma PRIORITY must be static (RM 9.8/3). Whether or not the subtype is static, an implementation must ensure that the pragma PRIORITY has no effect when the specified value lies outside the range defined for the subtype PRIORITY.

S5. ADDRESS is declared as a type. It cannot be a subtype of a predefined type (such as one of the predefined integer types). It might be declared as a composite type.

S6. If a package named SYSTEM is user-defined and compiled, the effect is implementation dependent, since an implementation can depend on declarations contained in the predefined package SYSTEM. Even if all the predefined declarations are given in the user-defined package, the effect can be unpredictable.

Changes from July 1982

S7. The named numbers for STORAGE_UNIT and MEMORY_SIZE are defined to have type *universal_integer*.

S8. The pragma SYSTEM_NAME has no effect if the argument is not one of the literals of type SYSTEM.NAME.

Changes from July 1980

S9. The package SYSTEM is now a predefined library unit (instead of being declared within STANDARD).

S10. The pragma SYSTEM is now called SYSTEM_NAME. The type SYSTEM_NAME is now called NAME.

S11. Type ADDRESS is now defined in package SYSTEM (and need not be an integer type).

S12. The following named numbers are now defined in SYSTEM: MAX_DIGITS, MAX_MANTISSA, FINE_DELTA, and TICK.

S13. The subtype PRIORITY is now defined in SYSTEM.

Test Objectives and Design Guidelines

T1. Check that the package SYSTEM is not declared within STANDARD and that it is a predefined library package.

Implementation Guideline: Check that a with clause is needed to access any of the declarations within SYSTEM, but that the with clause need only apply to the unit using a declaration from package SYSTEM. That is, declarations from package SYSTEM can be used in a package body or a subunit as long as a with clause has been given for the package specification or for some ancestor unit.

T2. Check that the type ADDRESS is declared within SYSTEM.

Implementation Guideline: Declare a variable of type ADDRESS and check if a value of this type can be assigned to it. Put the assignment in a separate test, since some implementations might make the type ADDRESS a limited private type.

T3. Check that the type NAME is declared within SYSTEM and that at least one value is in the type. Check that all the values of the type agree with the implementer's documentation.

Implementation Guideline: Use the IMAGE and VAL attributes to check the values.

T4. Check that the constant SYSTEM_NAME is declared within SYSTEM and that it has the appropriate value of type SYSTEM.NAME.

Check that the pragma SYSTEM_NAME can be used and that it gives the constant SYSTEM_NAME the specified value.

Implementation Guideline: Check for each value of type SYSTEM_NAME. Be sure the library is empty when making this check.

Check that the pragma SYSTEM_NAME is ignored if it does not appear at the start of a compilation.

Check that the pragma SYSTEM_NAME is ignored if its argument is not a value of type SYSTEM.NAME.

T5. Check that the constant STORAGE_UNIT is declared within SYSTEM and that the type of this value is *universal_integer*.

Check that the constant can be used in static expressions.

Check that STORAGE_UNIT has a nonzero value that agrees with the value specified by the implementer.

Check that the pragma STORAGE_UNIT can be used to define the value of the STORAGE_UNIT constant, at least for the values allowed by an implementation.

Check that the pragma STORAGE_UNIT has no effect if its argument is not an integer literal.

Implementation Guideline: The argument should be a static *universal_integer* expression.

Check that the pragma STORAGE_UNIT has no effect if it does not appear at the beginning of a compilation.

T6. Check that MEMORY_SIZE is declared within SYSTEM and that this constant has type *universal_integer*.

Check that MEMORY_SIZE can be used in static expressions.

Check that the pragma MEMORY_SIZE can be used to define the value of the MEMORY_SIZE constant.

Check that the pragma MEMORY_SIZE is ignored if its argument is not a numeric literal.

Implementation Guideline: The argument should be a static *universal_integer* expression.

Check that the pragma MEMORY_SIZE is ignored if it does not appear at the beginning of a compilation.

T7. Check that subtype PRIORITY is declared within SYSTEM (not within STANDARD) and that it is a subtype of the predefined INTEGER type.

Check that the range of subtype *PRIORITY* agrees with that specified for the implementation.

13.7.1 System-Dependent Named Numbers

Semantic Ramifications

S1. *MAX_INT* and *MIN_INT* give the largest (most positive) and smallest (most negative) value, respectively, of all the predefined integer types, excluding the type *universal_integer* (AI-00304). Since the predefined integer types are required to have symmetric ranges (with the possible exception of an additional negative value) (RM 3.5.4/7) and since integer type declarations must be accepted if their range lies within the range of a predefined integer type other than *universal_integer* (RM 3.5.4/6), the following declaration must always be accepted by an implementation:

```
type LARGEST_INT is range -SYSTEM.MAX_INT .. SYSTEM.MAX_INT;
```

S2. The value of *SYSTEM.TICK* must be static, i.e., it cannot have a value determined at run time since *TICK* is declared by a number declaration (RM 3.2/9 and RM 3.2.2/1). The staticness of *SYSTEM.TICK* can be determined by using *SYSTEM.TICK* in a context that requires a static value and in a context where the static value makes a program illegal. For example, if *SYSTEM.TICK* is 1.0, then the following case statement is illegal:

```
case TRUE is
  when (SYSTEM.TICK = 1.0) => ...
  when TRUE =>                -- illegal; duplicate choice
end case;
```

S3. The value of *TICK* has no required relation to the value of *DURATION'SMALL*. *DURATION'SMALL* must be no greater than 0.020 (RM 9.6/4), but *SYSTEM.TICK* can be larger or smaller than this value, since the value of *TICK* is determined by the accuracy of the clock.

S4. The value of *FINE_DELTA* is related to *MAX_MANTISSA* as follows:

```
T'FINE_DELTA = 2.0 ** (-T'MAX_MANTISSA)
```

Changes from July 1982

S5. There are no significant changes.

Changes from July 1980

S6. The definitions of *MAX_DIGITS*, *MAX_MANTISSA*, *FINE_DELTA*, *MIN_INT*, and *MAX_INT* are now given in this section.

Test Objectives and Design Guidelines

T1. Check that *MIN_INT* and *MAX_INT* have the values specified by the implementer, that these constants are both static and have the type *universal_integer*, and that no predefined integer type has values outside this range.

Implementation Guideline: Declare integer types that use these values.

T2. Check that *MAX_DIGITS* has the value specified by the implementer, and that this constant is static and has type *universal_integer*.

Check that a floating point declaration cannot have a specified digits value greater than *SYSTEM.MAX_DIGITS* (see IG 3.5.7/T1).

- T3. Check that MAX_MANTISSA has the value specified by the implementer and that the constant is both static and has type *universal_integer*.

Check that FINE_DELTA has the value specified by the implementer and that the constant is both static and has type *universal_real*.

Check that $\text{FINE_DELTA} = 2.0 ** (-\text{MAX_MANTISSA})$.

- T4. Check that TICK has the value specified by the implementer and that the constant is both static and has type *universal_real*.

Check that the precision of the clock is at least that specified for TICK.

13.7.2 Representation Attributes

Semantic Ramifications

S1. The value delivered by the ADDRESS attribute is implementation dependent and may not have any useful meaning in some cases. For example, since a subprogram is a program unit, the ADDRESS attribute can be applied to the name of an enumeration literal, predefined operator, or inlined subprogram. For an enumeration literal, there is no code associated with the "subprogram." For a predefined operator or inlined subprogram, there is either no code or there is no unique address associated with the code. In all such cases, an implementation must accept the attribute, but it can return any convenient value.

S2. When the ADDRESS attribute is applied to a formal parameter, the intention is to provide an address value suitable for accessing the actual parameter's value. When the actual parameter is passed by reference, the ADDRESS attribute should give the address of the actual parameter. When the parameter is passed by value, the address of the copy should be given. When the actual parameter is passed in a register or via a descriptor (e.g., an array might be passed using a descriptor that gives the lower and upper bounds for each dimension followed by a pointer to the contents of the array), the value returned by the address attribute might not be meaningful. An implementation can define additional attributes that give suitable values under these circumstances.

S3. The address of an object is also not always meaningful when the prefix denotes an object. For example, an optimizing compiler might not reserve space for a constant initialized with a static value. Similar, an object that is declared but never used need occupy no space. It is legal to specify the ADDRESS attribute for such objects, but the returned value will have no meaning.

S4. A context clause for SYSTEM need not apply (in the sense of RM 10.1.1/4-5) to a unit that uses the ADDRESS attribute. For example:

```
with SYSTEM;
package P is
    subtype MY_ADDRESS is SYSTEM.ADDRESS;
end P;

with P;
package Q is
    X : INTEGER := 0;
    Y : P.MY_ADDRESS := X'ADDRESS;           -- legal
```

The with clause for SYSTEM does not apply to package Q (and so it would be illegal to write SYSTEM.ADDRESS within package Q; see RM 10.1.1/5), but package Q does lie within the scope of the declaration for type ADDRESS (see IG 8.6/S), so the assignment operation is legal.

S5. The prefix of the attributes POSITION, FIRST_BIT, and LAST_BIT must have the form of a selected component and the prefix must name a record component. In particular, a name declared by a renaming declaration is not allowed as the prefix (AI-00238), and these attributes cannot be used when the prefix denotes an array component (AI-00362):

```

type REC is
  record
    C1 : INTEGER;
    C2 : STRING (1..5);
  end record;

type ARR_REC is array (1..5) of REC;

X : REC;
Y : STRING (1..5);
Z : ARR_REC;
C : INTEGER renames X.C1;

... X.C1'POSITION           -- legal
... C'POSITION              -- illegal; prefix not of form R.C
... Y(1)'POSITION           -- illegal; array component
... X.C2(I)'POSITION        -- illegal; array component
... Z(I).C1'POSITION        -- legal; Z(I) denotes a record

```

S6. The value returned by the SIZE attribute reflects the actual amount of space used to hold an object. An implementation decides whether the amount of space includes descriptive information, e.g., index subtype bounds for an array object.

S7. The size attribute can have the value zero if no space is allocated for an object, e.g., because an optimizing compiler has determined that the object need not be stored in memory. Each object having the same subtype need not occupy the same amount of space. For example, a record component having the subtype INTEGER range 1..7 might occupy only three bits, while an object that is not a component of an array or record might occupy a full word or a full byte; its size could be 8, 16, 32, or some other value.

S8. When the SIZE attribute is applied to a subtype, the value returned is the minimum number of bits the implementation will use to hold an object having that subtype. For example, consider the subtype INTEGER range 11..15. A signed representation will occupy at least 5 bits, an unsigned representation will occupy 4 bits, and a biased representation requires only 3 bits. If an implementation does not support unsigned or biased representations even for record components, the SIZE attribute for this subtype will return a value that is at least 5.

S9. RM 13.7.2/6 specifies how the name of a function is to be interpreted when it serves as the prefix of the SIZE or ADDRESS attributes, even though a function name is not allowed as the prefix of the SIZE attribute.

Changes from July 1982

S10. There are no significant changes.

Changes from July 1980

S11. The semantics of representation attributes are now specified in this section.

Legality Rules

L1. The prefix of the attribute ADDRESS must denote an object, a package, a subprogram, a

generic unit, a task type or a single task, a label, an entry, or an entry family (RM 13.7.2/2). (The prefix cannot denote a named number, an exception, or a type or subtype other than a task type, or an attribute.)

- L2. The prefix of the attribute SIZE must denote a type, a subtype, or an object (RM 13.7.2/4).
- L3. The prefix of the attributes POSITION, FIRST_BIT, and LAST_BIT must denote a selected component form of name whose prefix denotes a record object and whose selector denotes a component (RM 13.7.2/7, AI-000362, and AI-00258).
- L4. The prefix of the attribute STORAGE_SIZE must denote an access type or subtype, a task type, or a task object.

Test Objectives and Design Guidelines

- T1. Check that the prefix of the address attribute can denote an object, package, subprogram, generic unit, task type, single task, label, entry, or entry family.

Implementation Guideline: Include (in a separate test) a prefix that denotes one of the function attributes, SUCC, PRED, POS, VAL, IMAGE, and VALUE.

Check that the prefix of the address attribute cannot denote a named number, an exception, or a type or subtype other than a task type. Check that the prefix cannot be an attribute other than an attribute that denotes a function.

- T2. Check that the address attribute can be used in a compilation unit even if a with clause for package SYSTEM does not apply to the unit.

- T3. Check that the prefix of the attribute SIZE cannot be a subprogram, generic unit, package, named number, single task, label, entry, entry family, exception, or an attribute other than T'BASE.

Implementation Guideline: For the subprogram prefix, use a parameterless function name.

Check that the prefix of the attribute SIZE can be an object, a type (including a task type), or a subtype, and that a suitable value is returned.

Implementation Guideline: Check for all types, and for the prefix T'BASE.

- T4. Check that the prefix of the attributes POSITION, FIRST_BIT, and LAST_BIT cannot denote an array component and cannot be a name declared by a renaming declaration.

Check that the prefix of the attributes POSITION, FIRST_BIT, and LAST_BIT can be a record component and that suitable values are returned.

Implementation Guideline: For an implementation that accepts a record representation clause, check that the values specified in the clause are returned.

Implementation Guideline: Include a prefix that is a subcomponent of an array.

- T5. Check that the prefix of the attribute STORAGE_SIZE cannot be a subprogram.

Check that the prefix of the attribute STORAGE_SIZE can be an access type, a task type, a task object, or a single task.

Implementation Guideline: For an implementation that accepts length clauses specifying STORAGE_SIZE, check that suitable values are returned.

- T6. Check that the attribute ADDRESS is defined for objects having any type.

13.7.3 Representation Attributes of Real Types

Semantic Ramifications

- S1. MACHINE_ROUNDS is true if every predefined arithmetic operation "either returns an

exact result or performs rounding." "Performs rounding" is deliberately vague. In particular, if the result to be rounded is exactly halfway between possible rounded values, MACHINE_ROUNDINGS can be true regardless of which value is chosen. Otherwise, MACHINE_ROUNDINGS is true if and only if the closest value is chosen.

S2. MACHINE_OVERFLOWS is true if the underlying machine can detect overflow when it occurs and the occurrence of overflow causes an exception to be raised. The raising of overflow is to be understood in the context of optimizations allowed by RM 11.6/6. In particular, the expression $A*B/C$ need not raise any exception if the product is held in a double length register before the division is performed. In addition, MACHINE_OVERFLOWS can be true even if no exception is raised when a result lies outside the range of safe numbers but within the base type's range of values (see AI-00021; not yet decided).

S3. The predefined operations relevant to the value of MACHINE_OVERFLOWS include the conversion operations.

Changes from July 1982

S4. There are no significant changes.

Changes from July 1980

S5. Machine-dependent attributes of real types are now defined in this section.

Legality Rules

- L1. The prefix of MACHINE_ROUNDINGS and MACHINE_OVERFLOWS must denote a fixed or floating point type or subtype (RM 13.7.3/2).
- L2. The prefix of the attributes MACHINE_RADIX, MACHINE_MANTISSA, MACHINE_EMAX, and MACHINE_EMIN must denote a floating point type or subtype (RM 13.7.3/5).

Test Objectives and Design Guidelines

- T1. Check that the prefix of the MACHINE_ROUNDINGS and MACHINE_OVERFLOWS attribute cannot be an integer type.
- T2. Check that the prefix of the attributes MACHINE_RADIX, MACHINE_MANTISSA, MACHINE_EMAX, and MACHINE_EMIN cannot be a fixed point or integer type.
- T3. If the attribute MACHINE_ROUNDINGS is true, check that rounding is performed for addition, subtraction, multiplication, division, and conversion.
- T4. If the attribute MACHINE_OVERFLOWS is true, check that an exception is raised for overflow for addition, subtraction, multiplication, division, and conversion. If not true, check that no exception is raised for at least one of these operations.
- T5. Check that MACHINE_RADIX, MACHINE_MANTISSA, MACHINE_EMAX, and MACHINE_EMIN have the correct values for each value of digits accepted by an implementation.

Implementation Guideline: Check that appropriate values can be computed, e.g., that $(1/\text{MACHINE_RADIX}) \approx \text{MACHINE_MANTISSA} \times \text{MACHINE_EMIN}$, etc.

13.8 Machine Code Insertions

Semantic Ramifications

- S1. Support for machine code insertions is optional.
- S2. If machine code insertions are supported, a with clause naming the package MACHINE_CODE must apply (in the sense of RM 10.1.1/4-5) to the compilation unit containing code statements. For example, the following would be illegal:

```

with MACHINE_CODE;
package NEW_CODE is
    subtype TYPE_1 is MACHINE_CODE.TYPE_1;
end NEW_CODE;

with NEW_CODE;
procedure MACHINE is
begin
    NEW_CODE.TYPE_1 (LDI, 4000);      -- illegal
end MACHINE;

```

The code statement is illegal even though the scope of MACHINE_CODE.TYPE_1 includes the code statement (see IG 8.6/S); a with clause for the package MACHINE_CODE does not apply to the procedure MACHINE.

S3. The package MACHINE_CODE can contain as many type declarations as necessary. The storage layout for these types may be different from the layout normally used for similar type declarations.

S4. In principle, a program can declare a variable of any type declared in MACHINE_CODE.

Changes from July 1982

S5. A with clause for the package MACHINE_CODE must apply to the unit containing a machine code insertion.

S6. An implementation can impose further restrictions on the record aggregates allowed in code statements.

Changes from July 1980

S7. The record types needed for machine code insertions are defined to occur in a predefined library package called MACHINE_CODE.

Legality Rules

- L1. A code statement is only allowed in the body of a procedure (RM 13.8/3).
- L2. If a code statement is given in a procedure body, no declarative items are allowed in the declarative part of the body (except for use clauses), no exception handler is allowed, and no other form of statement is allowed (RM 13.8/3).
- L3. The base type of a record type used in a machine code statement must be declared in the predefined library package MACHINE_CODE (RM 13.8/4).
- L4. A with clause naming the predefined library package MACHINE_CODE must apply to any compilation unit containing a code statement (RM 13.8/4).
- L5. Additional restrictions may be imposed on the use of code statements by an implementation (RM 13.8/5).

Test Objectives and Design Guidelines

- T1. Check that if machine code insertions are supported, a with clause for the predefined package MACHINE_CODE must apply to the unit containing a code statement.

Implementation Guideline: Check that a with clause given for a package specification or a generic unit applies to the body and its subunits. Similarly, check a with clause for a subprogram declaration.

- T2. Check that a user-defined record type cannot be used in a code statement.

Implementation Guideline: If possible, use a record type that is identical to one declared in package MACHINE_CODE or use a type derived from a type declared in MACHINE_CODE.

- T3. Check that code statements are not allowed in the statement part of a package body, a task body, or a function.
- T4. If machine code insertions are supported, check that no declarative items (other than use clauses) are allowed in the procedure's declarative part.

Check that if a code statement is present, no exception handler is allowed.

Check that if a code statement is present, no other form of statement is allowed.

Implementation Guideline: Try a statement that assigns one formal parameter to another.

- T11. If machine code insertions are allowed, give a procedure body that uses code statements.

13.9 Interface to Other Languages

Semantic Ramifications

- S1. There are several interrelated issues concerning the pragma INTERFACE:

- There are many ways of declaring subprograms in Ada. In particular, an enumeration literal is a subprogram and so are predefined operators, derived subprograms, certain attributes (such as SUCC; see RM 3.5.5/8), and subprograms declared by a renaming declaration, a generic instantiation, and a subprogram body. A pragma INTERFACE applies (in the sense of RM 13.9/3) only to subprograms declared by an explicit subprogram declaration (AI-00410). It is illegal to supply a subprogram body for a subprogram to which the pragma applies (AI-00306).
- If a subprogram name in a pragma INTERFACE is overloaded but only some of the denoted subprograms are acceptable (see below), the pragma applies just to the acceptable subprograms (AI-00306).

Each of these issues is discussed in turn.

- S2. If a pragma INTERFACE applies to a particular subprogram and is accepted by the implementation, it is illegal to provide a body for the subprogram. For example:

```
package P is
  procedure R (B : INTEGER);
  pragma INTERFACE (XXX, R);
end P;

package body R is
  procedure R (B : INTEGER) is ... end R;      -- illegal
end R;
```

The pragma in the package specification is equivalent to promising that no body for the procedure will later be provided. If a body is provided, it must be rejected (AI-00306).

- S3. If a body is provided for a subprogram to which the pragma INTERFACE applies, it is irrelevant whether the body is provided before or after the occurrence of the pragma. The only relevant fact is whether the pragma names an acceptable language and the named subprogram has been declared earlier in the same declarative part or package specification:

```
declare
  procedure R (B : INTEGER);
```

```

    procedure R (B : INTEGER) is ... end R;      -- illegal
    pragma INTERFACE (XXX, R);
begin

```

Since procedure R is declared earlier in the same declarative part, the pragma applies to it and no body can be provided (AI-00306 and RM 13.9/3).

S4. If a body is required for a subprogram declared in a package specification, a package body that contains the subprogram body (or a stub for the body) must be provided. If a pragma INTERFACE applies to the subprogram, however, no subprogram body is needed (or allowed). Consequently, it may not be necessary to provide a package body. For example:

```

package P is
    procedure R (B : INTEGER);
    pragma INTERFACE (XXX, R);
end P;

```

No body is required for package P if the pragma is accepted.

S5. There is no harm in providing the pragma INTERFACE more than once for the same subprogram since the effect of the pragma is just to indicate that a body cannot be provided.

S6. RM 13.9/3 says that a subprogram name in the pragma INTERFACE is allowed to stand for several overloaded subprograms. If only some of the denoted subprograms are acceptable, e.g., if only some of the subprograms are declared earlier in the same declarative part or package specification, the pragma is obeyed only for those subprograms (AI-00306). For example:

```

procedure P (B : BOOLEAN);      -- P1
package R is
    procedure P (I : INTEGER);   -- P2
    pragma INTERFACE (XXX, P);

```

The pragma only applies to P2. Since the pragma applies to at least one subprogram, it is not ignored. Since it does not apply to P1, it is not illegal to provide a body for P1.

S7. A similar situation can arise in a package body:

```

package P is
    procedure R (I : INTEGER);
end P;

package body P is
    procedure R (B : BOOLEAN);      -- R1
    procedure R (I : INTEGER) is ... end R;    -- R2
    procedure R (F : FLOAT) is ... end R;      -- R3
    pragma INTERFACE (XXX, R);
end P;

```

The pragma only applies to procedure R1. It does not apply to R2 because R2 is declared in the package specification, not earlier in the same declarative part. The pragma does not apply to R3 because R3 is not declared by a subprogram declaration; it is declared by a body (AI-00410). Since the pragma does not apply to R2 or R3, these bodies are not illegal.

S8. For overloaded names, a pragma INTERFACE applies to a particular subprogram whether or not a body is provided. For example:

```

declare
  procedure R (B : BOOLEAN);
  procedure R (B : INTEGER);
  pragma INTERFACE (XXX, R);

  procedure R (B : INTEGER) is ... end R;      -- illegal
begin

```

The relative position of the pragma and the body is not relevant:

```

declare
  procedure R (B : BOOLEAN);
  procedure R (B : INTEGER);
  procedure R (B : INTEGER) is ... end R;      -- illegal
  pragma INTERFACE (XXX, R);

```

The body for procedure R is illegal because the pragma applies to the corresponding procedure declaration (AI-00306).

S9. The pragma only applies to subprograms declared by explicit subprogram declarations or renaming declarations (AI-00306). In particular, the pragma does not apply to enumeration literals, attributes that denote functions (such as T'SUCC; see RM 3.5.5/8), predefined operators, derived subprograms, or subprograms declared by a generic instantiation or a subprogram body (i.e., a subprogram body that is not preceded by a conforming subprogram declaration). If a name in the pragma is overloaded and denotes one or more of these forms of subprogram, the pragma is obeyed for (applies to) just those subprograms declared by the subprogram or renaming declarations. For example:

```

package P is
  procedure SPECIAL (B : BOOLEAN);
  procedure R (B : INTEGER);
  function R (L : INTEGER) return INTEGER renames "+";
  procedure R (B : BOOLEAN) renames SPECIAL;
  pragma INTERFACE (XXX, R);
end P;

```

The pragma applies to each subprogram denoted by R that was declared by an explicit subprogram declaration appearing earlier in package specification P. Only the function R does not satisfy this requirement, so the pragma does not apply to the subprogram denoted by function R.

S10. The pragma does not apply to generic formal subprograms because such subprograms are not declared by subprogram declarations (see RM 12.1/2) and because they do not occur in a package specification or declarative part.

Changes from July 1982

S11. There are no significant changes.

Changes from July 1980

S12. The pragma INTERFACE is allowed for library units.

Legality Rules

L1. A body cannot be supplied for a subprogram to which the pragma INTERFACE applies (RM 13.9/3 and AI-00306).

Test Objectives and Design Guidelines

- T1. Check that if a pragma INTERFACE is given in a package specification and applies to a subprogram declared in the package specification, no subprogram body can be given in the corresponding package body.

Check that no package body is required if the only reason for requiring a package body is for procedures declared in a package specification, and the pragma INTERFACE applies to each such procedure.

Check that if a pragma INTERFACE is given in a declarative part and applies to a subprogram declared earlier in the same declarative part, it is illegal to provide a body either before or after the pragma.

- T2. Check that the pragma INTERFACE is ignored if a name in the pragma does not denote any subprogram declared earlier in the same declarative part or package specification, or if the pragma is given after a library unit declaration, and one or more names in the pragma is not the name of the library unit.

- T3. Check that the pragma INTERFACE is ignored if a nonoverloaded subprogram name given in the pragma denotes an enumeration literal, an attribute that denotes a function (use the attributes SUCC, PRED, POS, VAL, IMAGE, and VALUE), a predefined operator, a derived subprogram, a subprogram declared by a renaming declaration, a subprogram declared by a generic instantiation, a subprogram declared by a subprogram body, a generic unit, or a generic formal subprogram.

Implementation Guideline: Check for both package specifications and declarative parts.

- T4. If a subprogram name in the pragma INTERFACE is overloaded,

- check that the pragma applies to every subprogram declared earlier in the same package specification or declarative part by an explicit subprogram declaration.

Implementation Guideline: For this part of the test, every overloaded subprogram declaration should be acceptable for the pragma.

- check that the pragma applies only to subprograms declared earlier in the same package specification or declarative part, and not to subprograms declared in outer declarative regions.

Implementation Guideline: If the pragma is given in a package body, check that it does not apply to subprograms declared in the package specification.

- check that the pragma applies only to subprograms that are not enumeration literals, predefined operators, derived subprograms, or subprograms declared by renaming declarations, generic instantiations, or subprogram bodies.

- T5. Check that the pragma INTERFACE can be given for a library subprogram.

13.10 Unchecked Programming**13.10.1 Unchecked Storage Deallocation****Semantic Ramifications**

- S1. Since the formal parameter of an instance of UNCHECKED_DEALLOCATION has mode in out, the actual parameter cannot be a constant having an access type.

S2. If objects X and Y designate the same object, then any attempt to evaluate Y for its value after the call FREE(X) is erroneous (AI-00356). In particular, consider:

```

declare
  type ACC_STR is access STRING;
  X : ACC_STR := new STRING' ("ABC");
  Y : ACC_STR := X;
  function FREE is UNCHECKED_DEALLOCATION (STRING, ACC_STR);
begin
  FREE (X);
  X := new STRING' ("DEF");
  if X = Y then                -- erroneous

```

X and Y initially designate the same object. The allocator then creates a new object to be designated by X's value. Since the value of Y is not altered by the FREE(X) call, Y still designates an object conceptually distinct from the object designated by X's new value. However, the effect of FREE is to allow the space designated by X's old value to be reused. In particular, the allocator might return the same access value as X had before, and if so, the comparison, X = Y, will evaluate to TRUE. However, this is not required; the comparison can equally well evaluate to FALSE. The effect of the FREE operation is to make erroneous any attempt to access Y's value; hence, the value of X = Y is not defined.

S3. An implementation must allow instantiations of UNCHECKED_DEALLOCATION with an unconstrained array or record type. This means that there should be no use of the OBJECT formal parameter that violates the rule given in RM 12.3.2/4 (see IG 12.3.2/S).

S4. If an instance of UNCHECKED_DEALLOCATION is applied to a value that designates a task object, the task continues to execute (if it is not already terminated).

Changes from July 1982

S5. There are no significant changes.

Changes from July 1980

S6. There are no significant changes.

Test Objectives and Design Guidelines

T1. Check that UNCHECKED_DEALLOCATION can be instantiated with any OBJECT type, and in particular, can be instantiated with an unconstrained array or record type.

Check that if FREE is called for a variable whose designated object is a task, the task continues to execute correctly (although it cannot be accessed).

13.10.2 Unchecked Type Conversions

Semantic Ramifications

S1. If the source and target types have different sizes, the effect of invoking an instance of unchecked conversion is implementation dependent.

S2. An implementation must allow instantiations of UNCHECKED_CONVERSION with an unconstrained array or record type. This means there should be no use of the formal parameters that violate the rule given in RM 12.3.2/4 (see IG 12.3.2/S).

Changes from July 1982

S3. Restrictions on the use of an instantiation of UNCHECKED_CONVERSION are to be described in Appendix F.

Changes from July 1980

S4. There are no significant changes.

Test Objectives and Design Guidelines

T1. Check that UNCHECKED_CONVERSION can be instantiated for a variety of types, including unconstrained array and record types.

Implementation Guideline: Use types whose declarations are structurally identical.

Chapter 14

Input-Output

14.1 External Files and File Objects

Semantic Ramifications

S1. The generic packages `SEQUENTIAL_IO` and `DIRECT_IO`, and the packages `TEXT_IO`, `IO_EXCEPTIONS`, and `LOW_LEVEL_IO` are predefined library units (RM C/22).

S2. The formal type of the generic packages `SEQUENTIAL_IO` and `DIRECT_IO` is not limited, so it is not possible to instantiate these packages with a task type. In general, the generic packages `SEQUENTIAL_IO` and `DIRECT_IO` cannot be instantiated with a limited type (RM 12.3.2/1-2).

S3. Access types, private types, and composite types with access or private component types may be specified as the `ELEMENT_TYPE` for `SEQUENTIAL_IO` or `DIRECT_IO`, but the effect of reading and writing values of such types is implementation-dependent. However, an implementation is free to raise `USE_ERROR` in response to calls to `CREATE` or `OPEN` of files of such types (RM 14.4/5).

S4. `SEQUENTIAL_IO` and `DIRECT_IO` can be instantiated for unconstrained arrays (including `STRING`) and records with discriminants (without default values) types. Whether these instantiations are legal depends on how the packages are implemented (RM 12.3.2/4). Within the generic body of `SEQUENTIAL_IO` or `DIRECT_IO` there might be an attempt to use `ELEMENT_TYPE` in a way that is not allowed for such actual parameters; such instantiations are then illegal. Since there is no requirement to allow such instantiations, their legality is implementation-dependent. Note that either all unconstrained arrays and records with discriminants can be instantiated or none of them can. In particular, it is implementation-dependent whether variable-length values (e.g., values defined by records with discriminants) can be read and written.

S5. The effect of sharing an external file is implementation-dependent. In particular, side effects on a property (index, size, ...) of one internal file can be caused by an I/O operation on another internal file associated with the same external file (RM 14.1/13), with one exception: if an external file is associated with two internal direct files, the current index is a property of each internal file (RM 14.2/4).

S6. The sequential and direct I/O packages must be instantiated for a specific data type before any I/O operations can be performed. An implementation may legally augment the I/O facilities by providing subprograms, predefined standard instantiations (e.g., for predefined types), and renaming declarations, suitably collected in library packages. These facilities, however, must not replace the standard packages defined in RM 14.

S7. The names of predefined input-output subprograms, exceptions, and types are not reserved. Such names are subject to renaming (RM 8.5), hiding (RM 8.3), and overloading (RM 6.6). Note also that such names must be prefixed by the name of the appropriate package instance, although this can usually be avoided by appropriate use clauses. However, when more than one I/O package is instantiated, the exceptions declared in `IO_EXCEPTIONS` must be named with expanded names, e.g.,

```
declare
  use TEXT_IO
```

```

package STRING_IO is new SEQUENTIAL_IO (STRING);
package INT_IO is new INTEGER_IO (INTEGER);
use STRING_IO, INT_IO;
INT_ITEM : INTEGER;

begin
  GET (INT_ITEM);          -- GET for INTEGER I/O
exception
  when DATA_ERROR => ...  -- illegal
end;
```

The reference to DATA_ERROR is illegal because DATA_ERROR is declared (by a renaming declaration) in both TEXT_IO and SEQUENTIAL_IO. Since an exception name is not overloadable, the use clause does not make the name directly visible (RM 8.4/6), even though SEQUENTIAL_IO.DATA_ERROR and TEXT_IO.DATA_ERROR both denote the same exception.

S8. Because an implementation can refuse to open or create a file for implementation-defined reasons, and because input-output on unopened files raises an exception, an implementation can "support" I/O by raising an exception for every call to one of the subprograms defined in RM 14. Only if a call is accepted (i.e., does not raise an exception) does RM 14 place some semantic restrictions on the behavior to be provided by an implementation.

S9. Although the word "file" suggests storage devices, the definition given in the first sentence of RM 14.1/1 is broad enough to include any I/O device (such as terminals, sensors, etc.).

S10. Although the RM states that "The values transferred for a given file must all be of one type" (RM 14.1/2), an implementation that allows more than one internal file to be associated with a given external file can allow values of different types to be transferred between the external file and the internal files.

Changes from July 1982

S11. There are no significant changes.

Changes from July 1980

S12. The package defining general user-level input-output has been changed to the two generic packages, DIRECT_IO and SEQUENTIAL_IO.

S13. The mode of a file is no longer associated with the file type but is given by an enumeration value.

Test Objectives and Design Guidelines

- T1. Check whether sequential and direct files can be instantiated for unconstrained array types and types with discriminants (see IG 14.2.1/T1).
- T2. Check whether an external file can be associated with more than one internal file (see IG 14.2.1/T7).
- T3. Check that the I/O exceptions must be named with expanded names if, for example, a use clause names both TEXT_IO and an instantiation of SEQUENTIAL_IO (see IG 8.4/T6).

14.2 Sequential and Direct Files

Semantic Ramifications

Changes from July 1982

- s1. There are no significant changes.

Changes from July 1980

- s2. The position in a direct file is expressed by an integer of type COUNT instead of FILE_INDEX.
- s3. The definition of *current size* has been changed. The current size is the index of the last element (previously called *end position*). The previous definition of *current size* is not associated with any term.
- s4. The current index is the index used for the next read or write on direct access I/O. Previously, there existed a separate current read position and current write position.

14.2.1 File Management

Semantic Ramifications

- s1. STORAGE_ERROR can be raised by any subprogram call if storage is not sufficient (RM 11.1/8). Nothing in RM 14 limits the exceptions raised by any subprogram to just those exceptions declared in IO_EXCEPTIONS.
- s2. Suppose that the name specified by CREATE includes the name of a directory to which a user only has read access, and that CREATE specifies that a file is to be opened for writing. NAME_ERROR could be raised on the basis that a writable external file cannot be identified (RM 14.2.1/4). It would be preferable, however, to raise USE_ERROR, on the basis that the environment does not support creation of the file for writing.
- s3. If the name specified by CREATE identifies an existing external file, then an implementation may either raise USE_ERROR, may overwrite the existing external file (RM 14.4/5), or may create a new file with the same name and a different version (if the operating system supports files with multiple versions).
- s4. OPEN can only be used successfully for external files that already exist. In particular, if an attempt is made to open a nonexistent external file for writing, then the exception NAME_ERROR must be raised; it is not permitted for such an OPEN to have the same effect as CREATE.
- s5. It is possible to CREATE a file with mode IN_FILE (RM 14.2.1/3). An implementation is allowed to raise USE_ERROR for such a call. If USE_ERROR is not raised, the effect of attempting to read such a file is not defined by the RM. In particular, creating a file in IN_FILE mode is not equivalent to:

```
CREATE (F1, OUT_FILE, "Name1");
RESET (F1, IN_FILE);
```

The effect of attempting to read a file created in IN_FILE mode is implementation-dependent, but this is not so if the file is created in OUT_FILE mode and then RESET to IN_FILE mode. In particular, for TEXT_IO, if SKIP_PAGE is called after RESET, no exception should be raised, since the TEXT_IO RESET operation would output a line terminator, a page terminator, and a file terminator (RM 14.3.1/4). Thus, SKIP_PAGE would read past the line terminator and the

page terminator without raising an exception (RM 14.3.4/17-19). If SKIP_PAGE were called after creating a file in IN_FILE mode, END_ERROR might be raised (since the file is empty), but such behavior is not required by the RM; the length and contents of such a file are undefined.

s6. The string returned by the NAME function should correspond to a full specification of the name. This will include the version number if the operating system supports multiple versions of an external file. It will also include the path to the external file if the operating system supports multiple directories, or if search rules are applied to identify an external file.

s7. Consider the following:

```
OPEN (F, OUT_FILE, "Test");
NAME_STR := new STRING' (NAME(1));
DELETE (F); -- (1)
OPEN (F1, OUT_FILE, NAME_STR.all); -- (2)
OPEN (F2, OUT_FILE, "Test"); -- (3)
```

If no exception is raised at (1), then the OPEN at (2) must raise NAME_ERROR, since the external file no longer exists (RM 14.2.1/7) and the result of the NAME function uniquely identifies an external file. If the environment allows multiple versions of a file, or if a search rule is applied to identify an external file belonging to one of several directories, then the OPEN in (3) will not raise NAME_ERROR if either an earlier version of the external file or a file in another directory satisfies the short name "Test."

s8. The RM's definitions of CREATE and DELETE imply that external files can be dynamically created and deleted during program execution. Some operating systems or Ada implementations may, however, require that all external files to be used by a program be declared to the operating system before program execution starts, and may allow deletion of an external file only after program execution has ended. The RM allows an implementation to always raise USE_ERROR when CREATE is called in such environments. However, the RM also allows (and users would probably prefer) CREATE with mode OUT_FILE to have the same effect as OPENing an existing external file in mode OUT_FILE.

s9. The correct semantic effects of dynamic deletion are determined by the effects on a subsequent CREATE and OPEN. If no exception is raised when DELETE is called, then:

- the external file must "no longer exist;"
- a subsequent CREATE of the same external file should succeed,
- a subsequent OPEN of the same external file must raise an exception.

If no exception is raised by DELETE, these semantic effects must be maintained even if the environment does not actually support dynamic deletion of files.

s10. The status of open external files at the end of the main program's execution is undefined. In particular, there is no requirement to flush the buffers associated with any files left open. Thus, what data is in such external files is implementation-dependent.

s11. The string returned by FORM must be a valid string (according to the implementation's Appendix F) for use in a subsequent OPEN of the same external file. That string should be useable for the CREATE of another external file, although, strictly speaking, the use of the string for the OPEN of another file is implementation-dependent and may raise the exception USE_ERROR.

Here is an example of a copy operation that uses the FORM string to copy a magnetic tape, preserving the tape format. The assumption is that the FORM string is used to define a tape format, and that when a file is opened for reading, its format is determined by the I/O system if it is not specified in the FORM argument. The example is based on tape I/O for MULTIOS.

```

procedure COPY_TAPE (OLD_VOL_NO, NEW_VOL_NO : STRING) is
  type ITEM_TYPE is array (1 .. 100) of INTEGER;
  type ACC_STR is access STRING;
  ITEM      : ITEM_TYPE;
  FORM_STR  : ACC_STR;
  F1, F2    : FILE_TYPE;
  package ITEM_IO is new SEQUENTIAL_IO (ITEM_TYPE);
  use ITEM_IO;
  function REPLACE (OLD_SUBSTR, NEW_SUBSTR, STR : STRING)
    return STRING is
    ...
  end REPLACE;
begin
  OPEN (F1, IN_FILE, "Name1", "-tape_ansi -vol " & OLD_VOL_NO);
  -- OPEN will determine formatting of tape volume.
  FORM_STR := new STRING' (FORM(F1));
  FORM_STR.all := REPLACE (OLD_VOL_NO, NEW_VOL_NO, FORM_STR.all);
  CREATE (F2, OUT_FILE, "Name2", FORM_STR.all);
  -- Same tape format will be used for F2.
  loop
    READ (F1, ITEM);
    WRITE (F2, ITEM);
  end loop;
exception
  when END_ERROR =>
    CLOSE (F1);
    CLOSE (F2);
end COPY_TAPE;

```

S12. If temporary external files are not given a name by an operating system, the NAME function will raise USE_ERROR (RM 14.4/5). Also, a temporary external file may cease to exist after a CLOSE. Even if the temporary external file does have a name, if it is deleted by the implementation after it is closed, then a subsequent OPEN using that name will raise NAME_ERROR (the external file does not exist). For example:

```

with TEXT_IO; use TEXT_IO;
procedure P is
  type ACC_STR is access STRING;
  NAME_STR : ACC_STR;
  F        : FILE_TYPE;
begin
  CREATE (F); -- (1)
  PUT (F, "Write to the file.");
  NAME_STR := new STRING' (NAME(F)); -- (2)
  CLOSE (F); -- (3)
  ...
  OPEN (FILE => F, NAME => NAME_STR.all); -- (4)
end P;

```

A temporary file is created at (1). USE_ERROR may be raised at (2) if the operating system does not assign names to temporary files. If the temporary file does have a name, then NAME_ERROR may be raised at (4) if closing the temporary file at (3) causes the temporary file to be deleted.

S13. If the CREATE procedure raises USE_ERROR or NAME_ERROR, the state of the system should not have been altered. For example, if the external file being created already exists, the external file should continue to exist if CREATE raises USE_ERROR because the external file cannot be replaced. If the file being created did not exist, then it should not exist after CREATE raised an exception.

S14. Incorrect use of the enumeration literal INOUT_FILE as the actual parameter associated with the formal parameter MODE in the routines CREATE, OPEN, or RESET instantiated for SEQUENTIAL_IO or TEXT_IO is detected at compile time since INOUT_FILE is not a value of the corresponding formal type. All other incorrect uses of a mode literal cause USE_ERROR to be raised. For example, if the internal file F is associated with an external sequential read only file, then:

```
RESET (F, INOUT_FILE); -- detected at compile time
RESET (F, OUT_FILE);  -- raises USE_ERROR if not allowed
```

S15. Ada's OPEN and CLOSE procedures are not necessarily equivalent to the operating system operations for opening and closing a file. The Ada OPEN and CLOSE procedures, respectively, associate an external file with an internal file and sever the association between the internal file and the external file. It is implementation-dependent when certain operating system file management operations occur with respect to the Ada OPEN and CLOSE procedures. Thus, calling OPEN with the same external file name and different internal file names need not raise an exception: the first OPEN can call the operating system procedures for accessing the external file; the second OPEN simply associates the "opened" external file with the file object given in the OPEN call. Such a use of OPEN causes an external file to be shared by more than one (internal) file object. (Note that STATUS_ERROR is raised only if OPEN is called for an *internal* file that is open, i.e., already associated with an external file.) If the same external file cannot be shared by two internal files, then USE_ERROR should be raised when an attempt is made to associate an internal file with an external file that is already open.

S16. When an external file contains heterogeneous record types — e.g., header records followed by detail records — it is necessary to open the same external file once for each record type. An implementation is not required to allow more than one internal file to be associated with the same external file, but such support can be extremely convenient. Such a capability allows varying length and different types of records to be read and written. Varying length I/O can also be supported by allowing instantiations for unconstrained array and record types.

S17. An implementation may allow some forms of file sharing and not others, e.g., opening the same external file for both reading and writing might not be supported.

S18. File sharing can occur in rather devious ways, e.g., by sharing a file among tasks.

```
task type T;
task body T is
  SHARED : FILE_TYPE;
begin
  OPEN (SHARED, "COMMON FILE");
  ...
end T;
TASK_FAMILY: array (1..10) of T;
```

S19. In general, it cannot be detected at compile time whether such a multiple association will occur; attempts to create such a multiple association must therefore be detected at run time (when OPEN is called) and raise an exception. If the attempt is not supported by the implementation (e.g., if an external file can be shared for reading but not for writing), the OPEN operation should raise USE_ERROR (RM 14.4/5).

S20. If several internal files are associated with one external file, it is very likely that the implementation will maintain separate data structures, independent of all internal files, to describe characteristics of the external files. For example, in order to implement DELETE, "a usage count" may be associated with the external file. Similarly, current size is a property of the external file (and therefore must be the same for all internal files associated with a given external file). On the other hand, the association between an internal file and an external file must be stored in the internal file's data structure. Similarly, the current index of a direct file is a property of an internal file (RM 14.2/4).

S21. If multiple associations are allowed, then care must be taken in implementing RESET and DELETE. In the case of RESET the following choices seem reasonable and are permissible:

- raise `USE_ERROR` (that is, disallow resetting an external file while it is associated with more than one internal file).
- have the effect of resetting the file propagated automatically to all internal files associated with the same external file (this would be a natural choice for a tape file).
- allow different internal files to have different views of the external file (this could be natural if the sequential external file resides on a random access device).

In the case of DELETE, the following are possible choices:

- raise `USE_ERROR`.
- automatically sever the association between the external file and all of its internal files (subsequent attempts to access the external file would then raise `STATUS_ERROR`).
- allow access to the external file using the other internal file associations, and delete the external file as soon as all these associations are severed. Any attempt to OPEN the file using the same file name must raise an exception. An attempt to CREATE a file using the same name should probably raise `USE_ERROR` if the file is not yet deleted.

S22. The association between an external file and an internal file is not necessarily severed automatically when the internal file ceases to exist, as in the following example.

```
package INT_IO is new SEQUENTIAL_IO (INTEGER);
procedure IRRESPONSIBLE_FATHER is
  SON : INT_IO.FILE_TYPE;
begin
  INT_IO.CREATE (SON, NAME => "HIS_NAME");
end IRRESPONSIBLE_FATHER;
```

The effect of calling `IRRESPONSIBLE_FATHER` is implementation-dependent with respect to the existence or the content of the external file ("HIS_NAME").

S23. If the input-output packages use tasking to provide file access synchronization, and instantiation of the packages, or declaration of a file object, or invocation of one of the package's procedures causes task objects (or task access types) to be declared, then all such objects (or, in the case of an access type, all task objects designated by objects of the access type) will depend (RM 9.4/1-4) on the innermost block, procedure body, or task body containing such declarations, and such a block, procedure, or task cannot terminate until all its dependent tasks have terminated. The proper mechanism for ensuring synchronous termination of dependent tasks is the selective wait with a terminate alternative (RM 9.7.1), which should be

used by the implementation. If no innermost block, procedure body, or task body exists, then the context must be that of a library package. The termination of tasks that depend on library packages does not affect the termination of the main program (and vice versa). Hence, if the input-output packages are instantiated as library packages, and tasks are activated that depend on these instantiations, the completion of the main program is not affected by the execution of these tasks (see IG 9.4/S).

S24. The number of external files that can be simultaneously open is implementation-defined.

S25. An implementation need not support the RESET operation (i.e., USE_ERROR can be raised). Note that the resetting of a file to a specific MODE could raise USE_ERROR if the MODE specified is not supported by the implementation.

Changes from July 1982

S26. OPEN raises the exception NAME_ERROR if no external file exists with the specified name.

S27. RESET raises the exception USE_ERROR if the environment does not support resetting to the specified mode for the external file.

Changes from July 1980

S28. The specifications of the subprograms CREATE, OPEN, CLOSE, DELETE, NAME, and IS_OPEN have changed significantly.

S29. IN_FILE, OUT_FILE, and INOUT_FILE are enumeration literals, whereas previously they were file types.

S30. The reasons for raising NAME_ERROR by CREATE have changed. Previously, NAME_ERROR was raised if creation of the external file was prohibited (e.g., the external file already existed). Now, it is raised if the string does not allow identification of an external file.

S31. The reasons for raising NAME_ERROR by OPEN have changed. Previously, NAME_ERROR was raised if no such external file existed or if its access was prohibited. Now, it is raised if the string does not allow identification of an external file; in particular, if no external file with the given name exists.

S32. The exception USE_ERROR for CREATE, OPEN, and DELETE has been added.

S33. Previously, DELETE would flag the file for deletion and the external file would cease to exist as soon as it was no longer associated with any internal file.

S34. DELETE no longer raises the exception NAME_ERROR; it now raises the exception STATUS_ERROR.

S35. The procedure RESET has been added.

S36. The function MODE has been added.

S37. The function FORM has been added.

Exception Conditions

The set of exceptions that can be raised is not limited to the following. Any predefined exception can be propagated by any of the subprograms defined in the input-output packages, as well as (unnameable) exceptions defined within the subprograms. All such additional exceptions must be mentioned in Appendix F for a given implementation.

CREATE Exceptions

- E1. **STATUS_ERROR** is raised if the internal file is already associated with an external file (RM 14.2.1/4).
- E2. **NAME_ERROR** is raised if the **NAME** string does not identify an external file (RM 14.2.1/4), e.g., because the name contains an illegal character, is too long, or is ill-formed in some way.
- E3. **USE_ERROR** is raised if the environment does not support the creation of an external file for the specified mode with the given name (RM 14.2.1/4), e.g., because the named external file already exists and cannot be overwritten (RM 14.4/5). **USE_ERROR** is also raised if the **FORM** string is not acceptable to the implementation (RM 14.2.1/4).
- E4. **DEVICE_ERROR** is raised if the external file cannot be created because of a malfunction of the underlying system (RM 14.4/6).
- E5. **STORAGE_ERROR** is raised if insufficient storage exists to permit creation of data structures associated with an internal file.

OPEN Exceptions

- E6. **STATUS_ERROR** is raised if the internal file is already associated with an external file (RM 14.2.1/7).
- E7. **NAME_ERROR** is raised if the **NAME** string does not identify an existing external file (RM 14.2.1/7), e.g., because the name is illegal or an external file with the specified name does not exist.
- E8. **USE_ERROR** is raised if the environment does not support the opening of an external file for the specified mode with the given name (RM 14.2.1/7). **USE_ERROR** is also raised if the **FORM** string is not acceptable to the implementation (RM 14.2.1/4).
- E9. **DEVICE_ERROR** is raised if the external file cannot be opened because of a malfunction of the underlying system (RM 14.4/6).
- E10. **STORAGE_ERROR** is raised if insufficient storage exists to permit creation of data structures associated with an internal file.

CLOSE Exceptions

- E11. **STATUS_ERROR** is raised if the internal file is not associated with an external file (RM 14.2.1/10).
- E12. **DEVICE_ERROR** is raised if the external file cannot be closed because of a malfunction of the underlying system (RM 14.4/6).

DELETE Exceptions

- E13. **STATUS_ERROR** is raised if the internal file is not associated with an external file (RM 14.2.1/13).
- E14. **USE_ERROR** is raised if deletion is not supported by the environment or the external file cannot be deleted (RM 14.2.1/13).
- E15. **DEVICE_ERROR** is raised if the external file cannot be deleted because of a malfunction of the underlying system (RM 14.4/6).

RESET Exceptions

- E16. STATUS_ERROR is raised if the internal file is not associated with an external file (RM 14.2.1/16).
- E17. USE_ERROR is raised if the environment does not allow the external file to be reset to the specified mode (RM 14.2.1/16).
- E18. MODE_ERROR is raised if RESET attempts to change the mode of a file that is serving as the current default input or default output file (RM 14.3.1/5).
- E19. DEVICE_ERROR is raised if the external file cannot be reset because of a malfunction of the underlying system (RM 14.4/6).

MODE Exceptions

- E20. STATUS_ERROR is raised if the internal file is not associated with an external file (RM 14.2.1/19).

NAME Exceptions

- E21. STATUS_ERROR is raised if the internal file is not associated with an external file (RM 14.2.1/22).

FORM Exceptions

- E22. STATUS_ERROR is raised if the internal file is not associated with an external file (RM 14.2.1/25).

Test Objectives and Design Guidelines

- T1. Check that SEQUENTIAL_IO and DIRECT_IO can be instantiated with boolean, integer, constrained string, enumeration, character, access, constrained array, record without discriminants or whose discriminants have defaults, fixed point, floating point, and private types.

Check whether SEQUENTIAL_IO or DIRECT_IO can be instantiated with an unconstrained array type (including STRING), or a record type with discriminants (without defaults).

Check that SEQUENTIAL_IO and DIRECT_IO cannot be instantiated for limited types (including task types, and composite types containing limited components).

- T2. Check to see that the exception STATUS_ERROR is raised for SEQUENTIAL_IO or DIRECT_IO when:

- an internal file is open and an attempt is made to CREATE or OPEN the internal file;
- an internal file is not open and an attempt is made to CLOSE, DELETE, or RESET a file, or to determine the MODE, NAME, or FORM of a file.

Check that NAME_ERROR is raised for SEQUENTIAL_IO and DIRECT_IO when the name string does not identify an external file for an OPEN or CREATE operation on files of type SEQUENTIAL_IO or DIRECT_IO.

Implementation Guideline: Use macro definitions for different implementations. Include a case where the external file does not already exist when an OPEN is attempted for an OUT_FILE.

Check that USE_ERROR is raised for SEQUENTIAL_IO and DIRECT_IO for operations not supported by the implementation.

Implementation Guideline: Check:

- for an OPEN of a file of mode IN_FILE, OUT_FILE, or INOUT_FILE;
- for a CREATE of a file of mode IN_FILE, OUT_FILE, or INOUT_FILE;
- for a RESET of a file to mode IN_FILE, OUT_FILE, or INOUT_FILE;
- when an implementation does not support RESET;
- when an implementation does not support DELETE.

T3. Check that IS_OPEN returns the proper values for files of type SEQUENTIAL_IO or DIRECT_IO in the following cases:

- after the internal file is declared but before it is open or created,
- following CREATE (both successful and unsuccessful),
- after a successful CLOSE,
- following OPEN (both successful and unsuccessful),
- after a RESET,
- after a DELETE.

T4. Check that a file can be closed and then re-opened for SEQUENTIAL_IO and DIRECT_IO.

Check that the name returned by NAME can be used in a subsequent OPEN for SEQUENTIAL_IO and DIRECT_IO.

Implementation Guideline: Close the file following the call to NAME, then re-open it. Otherwise the test would not be possible in an implementation that does not permit multiple internal files to be associated with an external file.

T5. Check that CREATE is permitted for a file of mode IN_FILE.

Implementation Guideline: Check whether USE_ERROR is raised.

T6. After a successful DELETE of an external file, check that the name of the external file can be used in a CREATE operation. Check that an OPEN operation using the file name raises NAME_ERROR.

T7. For SEQUENTIAL_IO or DIRECT_IO, check whether or not more than one internal file may be associated with the same external file.

Check whether two internal files with different ELEMENT_TYPES may be associated with the same external file.

Implementation Guideline: Check by creating one file, and then by opening the created external file with another internal file.

Implementation Guideline: Check when the external file is a permanent file or a temporary file.

Implementation Guideline: Determine whether an external file may be associated with both an internal direct file and an internal sequential file.

Implementation Guideline: Write a combination of different types to the same external file and check whether they can be read correctly. For DIRECT_IO, try reading them in a different order.

T8. Check that an external sequential and direct file specified by a null string name is not accessible after the completion of the main program, and that an external file specified by a non-null string name is accessible after the completion of the main program.

Implementation Guideline: Check whether a temporary external file is deleted after it is closed, and that the NAME function may raise USE_ERROR if a temporary file has no name.

Implementation Guideline: Use two tests: the first to create the sequential file and write data into it; the second, to read the data from it.

Check that if two files are created with null names specified, distinct temporary external files are created.

T9. Check that the default modes for CREATE are OUT_FILE for sequential and text files, and INOUT_FILE for direct files.

T10. For SEQUENTIAL_IO and DIRECT_IO, check that an external file ceases to exist after a successful DELETE, and that the NAME of the external file can be used in a CREATE operation.

Implementation Guideline: Attempt to OPEN the deleted file.

Check whether or not an external file associated with more than one internal file may be DELETED.

T11. For SEQUENTIAL_IO and DIRECT_IO, check:

- the file remains open after a RESET;
- a successful RESET will read and write its elements from the beginning of the file.

Implementation Guideline: For direct files, check that the current index is set to one after a successful RESET.

- a supplied MODE parameter in a RESET changes the mode of a given file. If no mode parameter is supplied, then the mode remains the same as it was before the RESET.

Check the effect of resetting an internal file upon other internal files accessing the same external file.

T12. Check that an instantiation of SEQUENTIAL_IO is needed before the following subprograms can be used: CREATE, CLOSE, OPEN, RESET, MODE, NAME, FORM, IS_OPEN, END_OF_FILE, DELETE.

Check that an instantiation of DIRECT_IO is needed before the following subprograms can be used: CREATE, CLOSE, OPEN, RESET, MODE, NAME, FORM, IS_OPEN, END_OF_FILE, DELETE, SET_INDEX, INDEX, SIZE.

Check that TEXT_IO gives access to the following subprograms: CREATE, CLOSE, OPEN, RESET, MODE, NAME, FORM, IS_OPEN, END_OF_FILE, DELETE, SET_INPUT, SET_OUTPUT, STANDARD_INPUT, STANDARD_OUTPUT, CURRENT_INPUT, CURRENT_OUTPUT, SET_LINE_LENGTH, SET_PAGE_LENGTH, LINE_LENGTH, PAGE_LENGTH, NEW_LINE, SKIP_LINE, END_OF_LINE, NEW_PAGE, SKIP_PAGE, END_OF_PAGE, SET_COL, SET_LINE, COL, LINE, PAGE, GET (for types CHARACTER and STRING), GET_LINE, PUT (for types CHARACTER and STRING), and PUT_LINE.

T13. Check that the subprograms CREATE, OPEN, CLOSE, DELETE, RESET, MODE, NAME, FORM, and IS_OPEN are available for SEQUENTIAL_IO and DIRECT_IO, and that the subprograms have the correct formal parameter names.

T14. Check that FILE_TYPE is limited.

T15. Check whether USE_ERROR is raised by the SEQUENTIAL_IO and DIRECT_IO CREATE when a named external file already exists and OUT_FILE is specified as the mode.

T16. Check that the mode INOUT_FILE is not allowed for SEQUENTIAL_IO.

T17. Determine, for SEQUENTIAL_IO and DIRECT_IO, the number of internal files an implementation can support.

T18. For SEQUENTIAL_IO and DIRECT_IO, check that a null string for FORM specifies the use of the default options of the implementation, as specified in Appendix F, for the external file.

- T19. For SEQUENTIAL_IO and DIRECT_IO, check that FORM returns the form string for the external file.
- T20. For SEQUENTIAL_IO and DIRECT_IO, check that if the implementation allows alternate forms of the name or form, that FORM and NAME return a full specification of the name and form.
- T21. For SEQUENTIAL_IO and DIRECT_IO, check whether CREATE, OPEN, RESET, and CLOSE operate on files of all supported data types (see IG 14.2.2/T and IG 14.2.4/T).

14.2.2 Sequential Input-Output

Semantic Ramifications

- S1. STORAGE_ERROR can be raised by any subprogram call if storage is not sufficient (RM 11.1/8). Nothing in RM 14 limits the exceptions raised by any subprogram to just those exceptions declared in IO_EXCEPTIONS.
- S2. If a READ operation for SEQUENTIAL_IO raises the exception DATA_ERROR, then the following READ operation reads the next element.
- S3. Consider the following:

```

CREATE (DATA_FILE, OUT_FILE, "DATA_FILE");
WRITE (DATA_FILE, DATA1);
WRITE (DATA_FILE, DATA2);
WRITE (DATA_FILE, DATA3);
RESET (DATA_FILE, IN_FILE);
READ (DATA_FILE, DATA4);
READ (DATA_FILE, DATA5);
READ (DATA_FILE, DATA6);
READ (DATA_FILE, DATA7);

```

The file will contain three elements. When the RESET operation is performed, the READ operation will read the element at the beginning of the file. The last READ operation raises END_ERROR since there are no more elements to read from the given file.

In the following example:

```

CREATE (DATA_FILE, OUT_FILE, "DATA_FILE");
WRITE (DATA_FILE, DATA1);
WRITE (DATA_FILE, DATA2);
WRITE (DATA_FILE, DATA3);
RESET (DATA_FILE);
WRITE (DATA_FILE, DATA4);
CLOSE (DATA_FILE);

```

DATA4 is written at the beginning of the file. The number of elements in the file can either be one element or three elements. Since RM 14.2.2/6 does not specify that the element written becomes the last element of the file, it is Implementation-dependent which element is the last element.

- S4. If an existing file is opened in OUT_FILE mode with a null FORM parameter, then the WRITE operation will be performed at the beginning of the file. An implementation may also allow the FORM parameter to be used to specify that writing starts at the end of the file.
- S5. In the definition of READ, the use of the phrase "can be interpreted as a value of the type" instead of "is a value of the type" is deliberate. Consider a main program:

```

procedure MAIN is
  type T is ...;      -- (1)

```

Type T is created at the instant at which elaboration of the declaration at (1) ends. In particular, different executions of the same program create different types. If the value read from a file were required to be of the proper type, files could not be used to exchange information between programs, including different executions of the same program.

s6. The RM says DATA_ERROR is raised if the value read "cannot be interpreted as a value (of the type used to instantiate SEQUENTIAL_IO)." The RM also says this check need not be performed by all implementations. Now consider the following examples:

```

package SEQ_WR is new SEQUENTIAL_IO (INTEGER);
package SEQ_RD is new SEQUENTIAL_IO (POSITIVE);
use SEQ_WR, SEQ_RD;
ELEM      : POSITIVE;
FILE_WR   : SEQ_WR.FILE_TYPE;
FILE_RD   : SEQ_RD.FILE_TYPE;
...
CREATE (FILE_WR, OUT_FILE, "TEMP");
WRITE (FILE_WR, -100);
WRITE (FILE_WR, 5);
CLOSE (FILE_WR);

OPEN (FILE_RD, IN_FILE, "TEMP");
READ (FILE_RD, ELEM);      -- exception raised here

```

An implementation could raise USE_ERROR for SEQ_RD.OPEN since the subtype of the elements in the file being opened is not compatible with the subtype used when writing the file (see RM 14.4/5). (This difference could be detected if SEQ_WR.CREATE stored appropriate information in the created file). Assuming, however, that USE_ERROR is not raised, SEQ_RD.READ can raise either DATA_ERROR or CONSTRAINT_ERROR or no exception at all. DATA_ERROR can be raised because the value being read, -100, although a value of POSITIVE's base type, is not a value of the subtype used to instantiate SEQUENTIAL_IO. Note that this check can be indicated in the body of READ as follows:

```

if VALUE_READ not in ELEMENT_TYPE then
  raise DATA_ERROR;
end if;

```

s7. Alternatively, the assignment to the formal parameter, ITEM, implies a check:

```

begin
  ITEM := VALUE_READ;
exception
  when CONSTRAINT_ERROR =>
    raise DATA_ERROR;
end;

```

s8. Although such statements can be written inside a generic body, an optimizing compiler is allowed to eliminate the code that actually makes the indicated checks, since VALUE_READ has the subtype ELEMENT_TYPE, and so does ITEM. An implementation is allowed to assume that a variable has a value that satisfies its subtype, and so:

```

VALUE_READ not in ELEMENT_TYPE

```

should always be identically TRUE. If a compiler does not exploit this optimization opportunity, however, it will be possible to raise DATA_ERROR.

S9. If DATA_ERROR is not raised by READ, CONSTRAINT_ERROR can be raised by the assignment of the formal variable to the actual variable, ELEM. However, the constraint check for actual parameters can be omitted by an implementation, since the actual and formal parameters have the same subtype. So it is possible that no exception will be raised at all.

S10. Now let us consider the situation for enumeration types:

```
type ENUM is (A, B, C, D, E);
for ENUM use (0, 2, 4, 6, 8);
E_ELEM : ENUM;
package SEQ_ENUM is new SEQUENTIAL_IO (ENUM);
ENUM_FILE : SEQ_ENUM.FILE_TYPE;
```

Presumably, when an implementation writes an ENUM value, it will actually write the value's integer representation. If so, then the following possibility arises:

```
OPEN (ENUM_FILE, IN_FILE, "TEMP");
READ (ENUM_FILE, E_ELEM);           -- (1)
READ (ENUM_FILE, E_ELEM);           -- (2)
```

Assuming that we are reading the same TEMP external file that was written for the example with integers, and assuming the OPEN is successful, then the first READ at (1) would be expected to raise DATA_ERROR, since the integer value read, -100, is less than the smallest value used to represent ENUM values. One would expect a cautious implementation to raise DATA_ERROR in this case (although the RM does not, of course, require that any exception be raised).

S11. Suppose that no exception is raised for the first READ and execution proceeds to the second READ at (2). This call reads the integer value 5. Because of the representation clause given for ENUM, there is no ENUM value whose representation is 5, and so DATA_ERROR can be raised. However, since 5 lies in the range of values used to represent ENUM values, the only way an implementation can detect that 5 does not represent an ENUM value is to check explicitly to see whether 5 is a valid representation for an ENUM value. Such a check may well be considered "too complex" (RM 14.2.2/4), and may be omitted by an implementation. Moreover, the check could not be written in Ada, since READ is a generic procedure. There is no way to generate all possible values of ELEMENT_TYPE within the generic unit SEQUENTIAL_IO, since ELEMENT_TYPE is private.

S12. For composite types, additional considerations come into play:

```
subtype STR10 is STRING(1 .. 10);

package SEQ_STR is new SEQUENTIAL_IO (STRING);
package SEQ_STR10 is new SEQUENTIAL_IO (STR10);

STR10_ELEM : STR10;
STR_FILE   : SEQ_STR.FILE_TYPE;
STR10_FILE : SEQ_STR10.FILE_TYPE;
...
CREATE (STR_FILE, OUT_FILE, "TEMP");
WRITE (STR_FILE, (1 .. 5 => 'A'));
WRITE (STR_FILE, (1 .. 11 => 'B'));
CLOSE (STR_FILE);

OPEN (STR10_FILE, IN_FILE, "TEMP");
READ (STR10_FILE, STR10_ELEM);      -- (1)
READ (STR10_FILE, STR10_ELEM);      -- (2)
```

Assuming the instantiation for SEQ_STR is allowed (see IG 14.1/S) and SEQ_STR10.OPEN does not raise USE_ERROR, the first READ at (1) should raise DATA_ERROR. DATA_ERROR could be raised relatively easily since the length of the string read does not equal the length required by the subtype used to instantiate SEQ_STR10. Since an implementation can usually detect the length of a record that is read, and since comparing the length against the expected length is a simple operation, this form of DATA_ERROR can be expected to be supported by many implementations (although such support is not, of course, required). The necessary check can be written as:

```
if ITEM'CONSTRAINED and ITEM'SIZE /= Size_of_Value_Read then
    raise DATA_ERROR;
end if;
```

If DATA_ERROR is not raised, then CONSTRAINT_ERROR will not be raised either when READ returns (even by a nonoptimizing compiler) since RM 6.4.1/9 explicitly states that no constraint check is performed for out parameters when a subprogram returns unless the parameter has a scalar or access type.

S13. The RM does not specify whether the bounds of an array are stored with the value:

```
STR20 : STRING (11 .. 20);
package SEQ_STR20 is new SEQUENTIAL_IO (STR20);
```

If we write to a file using SEQ_STR10 and then read from the file using SEQ_STR20, an implementation could raise DATA_ERROR because the bounds of the written values do not equal the bounds required for the values that are read, or the implementation could simply confirm that the required number of values is read, and then the assignment to the formal parameter of SEQ_STR20.READ will implicitly change the bounds without raising an exception. If the bounds of arrays are written and read, then the appropriate check can be written as

```
if VALUE_READ not in ELEMENT_TYPE then
    raise DATA_ERROR;
end if;
```

Alternatively:

```
begin
    ITEM := ELEMENT_TYPE' (VALUE_READ);
exception
    when CONSTRAINT_ERROR =>
        raise DATA_ERROR;
end;
```

Note that without qualifying VALUE_READ, CONSTRAINT_ERROR will not be raised as long as VALUE_READ'LENGTH = ITEM'LENGTH. And, of course, even if the above block is written in the template for READ, an optimizing compiler is allowed to omit the check implied by the type qualification, since VALUE_READ must have the subtype ELEMENT_TYPE.

S14. Similar reasoning applies to the reading and writing of variant records:

```
type VREC (D : INTEGER) is
    record ... end record;
subtype VREC_3 is VREC(3);

package SEQ_VREC is new SEQUENTIAL_IO (VREC);
package SEQ_VREC_3 is new SEQUENTIAL_IO (VREC_3);
```

```
VREC_ELEM   : VREC(3);
VREC_FILE   : SEQ_VREC.FILE_TYPE;
VREC_3_FILE : SEQ_VREC_3.FILE_TYPE;
```

```
CREATE (VREC_FILE, OUT_FILE, "TEMP");
WRITE (VREC_FILE, (1, ...));
CLOSE (VREC_FILE);
```

```
OPEN (VREC_3_FILE, IN_FILE, "TEMP");
READ (VREC_3_FILE, VREC_ELEM);
```

If an attempt is made to check the length or the subtype of the value read, DATA_ERROR can be raised.

S15. The RM does not specify in general how an implementation should cope with access contentions. For example, the RM does not state that a WRITE should not be initiated during another WRITE, or during a READ, etc. This allows simple implementations of the I/O packages (for applications not requiring concurrent access to files) to be interchangeable with more elaborate implementations.

Changes from July 1982

S16. WRITE raises the exception USE_ERROR if the capacity of the external file is exceeded.

Changes from July 1980

S17. The specification of the subroutines READ, WRITE, END_OF_FILE have changed significantly.

S18. IN_FILE and OUT_FILE are enumeration literals, whereas before they were file types.

S19. The exception MODE_ERROR for READ, WRITE, and END_OF_FILE has been added.

Exception Conditions

The set of exceptions that can be raised is not limited to the following. Any predefined exception can be propagated by any of the subprograms defined in the input-output packages, as well as (unnamable) exceptions defined within the subprograms. All such additional exceptions must be mentioned in Appendix F for a given implementation.

READ Exceptions

- E1. STATUS_ERROR is raised if the file is not associated with an external file (RM 14.2.2/1).
- E2. MODE_ERROR is raised if the mode is not IN_FILE (RM 14.2.2/4).
- E3. DEVICE_ERROR is raised if an external file cannot be read because of a malfunction of the underlying system (RM 14.4/6).
- E4. END_ERROR is raised if no more elements can be read from the given file (RM 14.2.2/4).
- E5. DATA_ERROR is raised if the element read cannot be interpreted as a value of the type ELEMENT_TYPE (RM 14.2.2/4). An implementation is allowed to omit this check if performing the check is too complex.

WRITE Exceptions

- E6. STATUS_ERROR is raised if the file is not associated with an external file (RM 14.2.2/1).
- E7. MODE_ERROR is raised if the mode is not OUT_FILE (RM 14.2.2/7).
- E8. USE_ERROR is raised if the capacity of the external file is exceeded (RM 14.2.2/7).
- E9. DEVICE_ERROR is raised if the attempt to write to an external file cannot be completed because of a malfunction of the underlying system (RM 14.4/6).

END_OF_FILE Exceptions

- E10. STATUS_ERROR is raised if the file is not associated with an external file (RM 14.2.2/1).
- E11. MODE_ERROR is raised if the mode is not IN_FILE (RM 14.2.2/10).
- E12. DEVICE_ERROR is raised if the check for end of file cannot be completed because of a malfunction of the underlying system (RM 14.4/6).

Test Objectives and Design Guidelines

- T1. Check that READ, WRITE, and END_OF_FILE are supported for sequential files with ELEMENT_TYPES integer, boolean, access, enumeration, constrained array, constrained record, float, fixed, and private.

Implementation Guideline: Check that data written can be read. Try reading the data after a RESET or after closing and opening the file.

Check whether READ, WRITE, and END_OF_FILE are supported for an unconstrained record type with discriminants (with or without default values) and for an unconstrained array type.

- T2. Check that READ, WRITE, and END_OF_FILE raise STATUS_ERROR when applied to a non-opened sequential file. Check that USE_ERROR is not permitted for this condition.
- T3. Check that WRITE raises the exception USE_ERROR if the capacity of the external file is exceeded.

To the extent possible and practical, check that hardware malfunctioning causes DEVICE_ERROR to be raised by READ and WRITE.

- T4. Check that WRITE raises MODE_ERROR for sequential files of mode IN_FILE.

Check that READ and END_OF_FILE raise MODE_ERROR for sequential files of mode OUT_FILE.

- T5. Check whether READ for a sequential file raises DATA_ERROR when an invalid element is read and that reading can continue after the exception has been handled.

Implementation Guideline: Check whether CONSTRAINT_ERROR is raised instead of DATA_ERROR for a scalar type.

Implementation Guideline: Write to the file elements of one type and read them back as elements of another type.

Implementation Guideline: Check for value not in subtype (of equal and not equal range length), and for represented value not valid for enumeration type.

- T6. Check that READ for a sequential file raises END_ERROR when there are no more elements that can be read from the given file.

Check that END_OF_FILE detects the end of the sequential file correctly.

- T7. Check that INOUT_FILE is not an acceptable mode for sequential files.

- T8. Check that data can be overwritten in the sequential file and the correct values can later be read.

Check whether overwriting truncates the file to the last element written.

- T9. Check that a sequential file instantiation for type CHARACTER can read and write every ASCII character.

- T10. Check that Fortran-like READ and WRITE statements are illegal.

14.2.3 Specification of the Package Sequential_IO

The implications of the SEQUENTIAL_IO package have been discussed in IG 14.2.2.

14.2.4 Direct Input-Output

Semantic Ramifications

- S1. STORAGE_ERROR can be raised by any subprogram call if storage is not sufficient (RM 11.1/8). Nothing in RM 14 limits the exceptions raised by any subprograms to just those exceptions declared in IO_EXCEPTIONS.

- S2. If a READ operation for DIRECT_IO raises the exception DATA_ERROR, then the following READ operation reads the next element. In other words, the current index is updated even if the exception DATA_ERROR is raised.

- S3. For a direct file, if the current index is positioned where no element exists, the effect of the READ operation is implementation-dependent. The READ operation can raise the exception DATA_ERROR, or it can attempt to interpret the garbage as a value of the given type ELEMENT_TYPE, e.g.:

```
CREATE (F, OUT_FILE, "F_DIR");
WRITE (F, ITEM, 10);
RESET (F, IN_FILE);    -- Next READ will read first item.
READ (F, ITEM);        -- DATA_ERROR is raised or the undefined element
                        -- is interpreted as a value of type ITEM_TYPE.
```

- S4. The size of the file is increased by a WRITE operation, not by a SET_INDEX operation. If the size of the file F is 10, then:

```
SET_INDEX (F, 15);    -- SIZE is still 10
                        -- INDEX is now 15
WRITE (F, ITEM);      -- SIZE is now 15
                        -- INDEX is now 16
```

- S5. Consider the following:

```
CREATE (DATA_FILE, OUT_FILE, "DATA_FILE");
WRITE (DATA_FILE, DATA1);
WRITE (DATA_FILE, DATA2);
WRITE (DATA_FILE, DATA3);
RESET (DATA_FILE, IN_FILE);
READ (DATA_FILE, DATA4);
READ (DATA_FILE, DATA5);
```

```

READ (DATA_FILE, DATA6);
READ (DATA_FILE, DATA7);

```

The file will contain three elements. When the RESET operation is performed, the READ operation will read the element at the beginning of the file. The last READ operation raises END_ERROR since the current index exceeds the size of the given external file.

In the following example:

```

CREATE (DATA_FILE, OUT_FILE, "DATA_FILE");
WRITE (DATA_FILE, DATA1);
WRITE (DATA_FILE, DATA2);
WRITE (DATA_FILE, DATA3);
RESET (DATA_FILE);
WRITE (DATA_FILE, DATA4);
CLOSE (DATA_FILE);

```

DATA4 is written at the beginning of the file. The number of elements in the file is three elements.

Furthermore, if an existing file is opened in OUT_FILE mode with a null FORM parameter, the WRITE operation will occur at the beginning of the file. An implementation may also allow the FORM parameter to be used to specify that writing starts at the end of the file.

S6. A program can attempt to write more than COUNT'LAST file elements. Such an attempt need not raise an exception, but an implementation is free to raise USE_ERROR if an attempt is made to write more than COUNT'LAST elements.

S7. In the definition of READ, the use of the phrase "can be interpreted as a value of the type" instead of "is a value of the type" is deliberate. Consider a main program:

```

procedure MAIN is
  type T is ...;      -- (1)

```

Type T is created at the instant at which elaboration of the declaration at (1) ends. In particular, different executions of the same program create different types. If the value read from a file were required to be of the proper type, files could not be used to exchange information between programs, including different executions of the same program.

S8. Note that the RM does not specify in general how an implementation should cope with access contentions. For example, the RM does not state that a WRITE should not be initiated during another WRITE, or during a READ, etc. This allows simple implementations of the I/O packages (for application not requiring concurrent access to files) to be interchangeable with more elaborate implementations.

S9. See IG 14.2.2/S for a discussion of when DATA_ERROR can be raised.

Changes from July 1982

S10. WRITE raises the exception USE_ERROR if the capacity of the external file is exceeded.

S11. SIZE returns the current size of the external file (instead of the number of elements in the external file) that is associated with the given file.

Changes from July 1990

S12. The specification of the subprograms READ, WRITE, SIZE, and END_OF_FILE have changed significantly.

S13. IN_FILE, OUT_FILE, and INOUT_FILE are enumeration literals, whereas before they were file types.

- S14. The exception `MODE_ERROR` for `READ`, `WRITE`, and `END_OF_FILE` has been added.
- S15. The definition of current size has been changed.
- S16. The procedure `SET_INDEX` replaces the subprograms `SET_READ`, `RESET_READ`, `SET_WRITE`, and `RESET_WRITE`.
- S17. The function `INDEX` replaces the subprograms `NEXT_READ` and `NEXT_WRITE`.
- S18. The function `LAST` has been eliminated.

Exception Conditions

The set of exceptions that can be raised is not limited to the following. Any predefined exception can be propagated by any of the subprograms defined in the input-output packages, as well as (unnamed) exceptions defined within the subprograms. All such additional exceptions must be mentioned in Appendix F for a given implementation.

READ Exceptions

- E1. `STATUS_ERROR` is raised if the file is not associated with an external file (RM 14.2.4/1).
- E2. `MODE_ERROR` is raised if the mode of the given file is `OUT_FILE` (RM 14.2.4/4).
- E3. `DEVICE_ERROR` is raised if the read operation cannot be completed because of a malfunction of the underlying system (RM 14.4/6).
- E4. `END_ERROR` is raised if the index to be used exceeds the size of the external file (RM 14.2.4/4).
- E5. `DATA_ERROR` is raised if the element read cannot be interpreted as a value of the type `ELEMENT_TYPE` (RM 14.2.4/4). Note that an implementation is allowed to omit this check if performing this check is too complex.

WRITE Exceptions

- E6. `STATUS_ERROR` is raised if the file is not associated with an external file (RM 14.2.4/1).
- E7. `MODE_ERROR` is raised if the mode of the given file is `IN_FILE` (RM 14.2.4/7).
- E8. `USE_ERROR` is raised if the capacity of the external file is exceeded (RM 14.2.4/7).
- E9. `DEVICE_ERROR` is raised if the write operation cannot be completed because of a malfunction of the underlying system (RM 14.4/6).

SET_INDEX Exceptions

- E10. `STATUS_ERROR` is raised if the file is not associated with an external file (RM 14.2.4/1).
- E11. `DEVICE_ERROR` is raised if this operation cannot be completed because of a malfunction of the underlying system (RM 14.4/6).

INDEX Exceptions

- E12. `STATUS_ERROR` is raised if the file is not associated with an external file (RM 14.2.4/1).

SIZE Exceptions

- E13. `STATUS_ERROR` is raised if the file is not associated with an external file (RM 14.2.4/1).

END_OF_FILE Exceptions

E14. **STATUS_ERROR** is raised if the file is not associated with an external file (RM 14.2.4/1).

E15. **MODE_ERROR** is raised if the mode of the given file is **OUT_FILE** (RM 14.2.4/16).

E16. **DEVICE_ERROR** is raised if the check for end of file cannot be completed because of a malfunction of the underlying system (RM 14.4/6).

Test Objectives and Design Guidelines

- T1. Check that **READ** (with and without parameter **FROM**), **WRITE** (with and without parameter **TO**), **SET_INDEX**, **INDEX**, **SIZE**, and **END_OF_FILE** are supported for direct files with **ELEMENT_TYPES**, character, integer, boolean, enumeration, access, constrained array, record without discriminants, floating point, fixed point, and private.

Check whether **READ** (with or without parameter **FROM**), **WRITE** (with or without parameter **TO**), **SET_INDEX**, **INDEX**, **SIZE**, and **END_OF_FILE** are supported for an unconstrained array type and a record type with discriminants (with or without defaults).

Check that data written into a direct file can be read correctly.

Implementation Guideline: Try reading the data after a **RESET**, after closing and opening the file again, or after a **WRITE** while in **INOUT_FILE** mode.

Implementation Guideline: Try writing the data in random positions (leaving gaps so that the data in the direct file are not contiguous) and then reading them in a different order.

Check that data can be overwritten in the direct file and the correct values can later be read.

Implementation Guideline: Try while in **INOUT_FILE** mode, and by writing in mode **OUT_FILE** and then resetting to **IN_FILE**.

Check that rewriting an element does not change the size of the file.

- T2. Check that **READ**, **WRITE**, **INDEX**, **SET_INDEX**, **SIZE**, and **END_OF_FILE** raise **STATUS_ERROR** when applied to a non-open direct file. **USE_ERROR** is not permitted.

- T3. Check that **WRITE** raises the exception **USE_ERROR** if the capacity of the external file is exceeded.

To the extent possible and practical, check that hardware malfunctioning causes **DEVICE_ERROR** to be raised by **READ** and **WRITE**.

- T4. Check that **READ** raises **MODE_ERROR** when the current mode is **OUT_FILE**.

- T5. Check whether **READ** for **DIRECT_IO** raises **DATA_ERROR** when the current index corresponds to an undefined element, or when a value read does not belong to **ELEMENT_TYPE**. Check that reading can continue after the exception has been handled.

Check that **READ** raises **END_ERROR** when the current index is greater than the current size.

Check that **END_OF_FILE** detects the end of a direct file correctly.

- T6. For a direct access file of mode **IN_FILE** and mode **INOUT_FILE**, check that, after a **READ**, the current read position is incremented by one.

- T7. Check that **WRITE** raises **MODE_ERROR** when the current mode is **IN_FILE**.

- T8. Check that **WRITE** does not cause an exception when the **TO** parameter is greater than the file size, or when the current index exceeds the size.

- T9. For direct access files of mode **OUT_FILE** and mode **INOUT_FILE**, check that a **WRITE** to

a position greater than the current size causes the current index and the file size to be incremented.

Check that the size of the file is correctly returned by SIZE.

T10. Check that END_OF_FILE raises MODE_ERROR when the current mode is OUT_FILE.

T11. Check that INDEX returns the correct index position in the direct file.

Implementation Guideline: Use INDEX after several WRITE operations or after WRITE using the TO parameter.

Check that SET_INDEX correctly sets the index position in the direct file.

T13. Check that Fortran-like READ and WRITE statements are illegal.

14.2.5 Specification of the Package Direct_IO

The implications of the DIRECT_IO package have been discussed in IG 14.2.4.

14.3 Text Input-Output

Semantic Ramifications

S1. The RM prescribes that the standard input and output files are "associated with two ... external files" (RM 14.3/5) (emphasis added). This does not preclude the use of an interactive device for both input and output, since a *file* is a sequence of characters; the relationship between the files and the physical actions that take place on the device is not prescribed by the language.

S2. The current column, line, and page number, and also the maximum line and page length, are all properties of an internal file. For example, consider the following:

```
SOURCE, SINK : FILE_TYPE;
...
OPEN (SOURCE, IN_FILE, "PIPE_FILE");
OPEN (SINK, OUT_FILE, "PIPE_FILE");  -- if allowed by implementation
```

The call

```
NEW_LINE (SINK);
```

does not affect the line and column number of SOURCE.

S3. Although NEW_LINE, SET_COL, etc. affect properties of internal files (in particular, the values returned by LINE, COL, and PAGE), the NEW_LINE, SET_COL, etc. procedures take a file value as an in parameter. Hence, the current column, line, and page position cannot be stored directly in a file value, but must be stored indirectly. Moreover, although a text file is a limited private type (see RM 14.3.10) whose implementation is not dictated by the RM, the full declaration of FILE_TYPE must either be an access or a composite type to ensure that file objects never have undefined values:

```
procedure P is
  F : FILE_TYPE;
begin
  SET_INPUT(F);  -- F is an in parameter
```

The call to SET_INPUT is defined to raise STATUS_ERROR (RM 14.3.2/4), i.e., the program is

not erroneous even though no value has apparently been given to *F* (see RM 3.2.1/18 and IG 3.2.1/S). The only way *F* can be referenced nonerroneously is if the full declaration of *FILE_TYPE* declares an access type or as a composite type. It would be incorrect to implement *FILE_TYPE* as a scalar type, e.g., an integer type whose value serves as an index into an array of file control blocks.

s4. Use of a text file as a default file is a property of the internal file, since an attempt to change the mode of the file raises an exception (see IG 14.3.1/S). Hence, an implementation must keep track of whether a text file is being used as a default file.

s5. The physical representation of page and line terminators is implementation-dependent (RM 14.3/7). In particular, an implementation could choose to allow a single ASCII character to represent a sequence of terminators, e.g., the ASCII FF (form feed) character could represent the sequence, <line terminator>, <page terminator>.

s6. The RM permits text file output to fixed record media, e.g., cards. The effect of such output may be to pad lines with spaces, since nothing in the RM forbids such padding.

Changes from July 1982

s7. A line terminator is explicitly defined to have a column number.

Changes from July 1980

s8. Files are divided into pages as well as lines.

s9. *OPEN*, *CLOSE*, *CREATE*, *DELETE*, *RESET*, *MODE*, *NAME*, *FORM*, and *IS_OPEN* procedures are explicitly declared within the *TEXT_IO* package instead of using the *INPUT_OUTPUT* procedures for files of type *CHARACTER*.

s10. *GET* and *PUT* procedures are defined for reading and writing from a parameter of type *STRING*.

Test Objectives and Design Guidelines

T1. Check that the package *TEXT_IO* exists and that *READ* and *WRITE* are not available.

T2. Check that

- *FILE_TYPE* is visible and is limited private;
- *FILE_MODE* is visible and has values *IN_FILE* and *OUT_FILE*, but not *INOUT_FILE*;
- *COUNT* is visible, *COUNT'FIRST* = 0, and *COUNT'LAST* has an appropriate implementation-defined value;
- *POSITIVE_COUNT* is a subtype of *COUNT*, *POSITIVE_COUNT'FIRST* = 1, and *POSITIVE_COUNT'LAST* = *COUNT'LAST*;
- *FIELD* (RM 14.3.7/2) is a subtype of *INTEGER*, *FIELD'FIRST* = 0, and *FIELD'LAST* has an appropriate implementation-defined value;
- *NUMBER_BASE* (RM 14.3.7/2) is a subtype of *INTEGER*, with *NUMBER_BASE'FIRST* = 2 and *NUMBER_BASE'LAST* = 16;
- *UNBOUNDED* is a constant of type *COUNT* with value zero.

T3. Check that the default input and output files are distinct external files, e.g., check that they have distinct names.

Implementation Guideline: Use the *NAME* function to determine the default file names.

14.3.1 File Management

Semantic Ramifications

S1. The following example shows how RESET can raise MODE_ERROR when attempting to reset the mode of a file currently serving as a default input or default output file:

```
CREATE (FT, OUT_FILE);
SET_OUTPUT (FT);
RESET (FT, IN_FILE);      -- MODE_ERROR raised
```

S2. Note that STORAGE_ERROR can be raised by any subprogram call if storage is not sufficient (RM 11.1/8). Nothing in RM 14 limits the exceptions raised by any subprogram to just those exceptions declared in IO_EXCEPTIONS.

S3. Suppose the name specified by CREATE includes the name of a directory to which a user only has read access, and CREATE specifies that a file is to be opened for writing. NAME_ERROR could be raised on the basis that a writable external file cannot be identified (RM 14.2.1/4). It would be preferable, however, to raise USE_ERROR, on the basis that the environment does not support creation of a file for writing.

S4. If the name specified by CREATE identifies an existing external file, an implementation may either raise USE_ERROR or may overwrite the existing file (RM 14.4/5).

S5. Note that OPEN can only be successfully used for files that already exist. If a nonexistent file is opened for writing, the exception NAME_ERROR must be raised; it is not permitted for such an OPEN to have the same effect as CREATE.

S6. Note that resetting a file to OUT_FILE always resets the line and page lengths to UNBOUNDED (RM 14.3.1/4), even if the previous mode was OUT_FILE.

S7. Implementations may require that some, or that all, external text files have a bounded line length, but the RM specifies that all text files initially have an unbounded line length (RM 14.3.1/2). Although it would be allowed to raise USE_ERROR when attempting to OPEN or CREATE files required to have bounded line lengths, this would not be very helpful. Another approach is to raise USE_ERROR if PUT attempts to exceed the line length required by the external file (RM 14.4/5). Note that any attempt to explicitly set the line length for such files must raise USE_ERROR if the specified length is UNBOUNDED (RM 14.3.3/5). Hence, an attempt to set the line length to its current value can raise USE_ERROR if LINE_LENGTH = UNBOUNDED.

S8. It is possible to call CREATE with mode IN_FILE (RM 14.2.1/3). An implementation is allowed to raise USE_ERROR for such a call. If USE_ERROR is not raised, then the effect of attempting to read such a file is not defined by the RM.

S9. The RM's definitions of CREATE and DELETE imply that files can be dynamically created and deleted during program execution. Some operating systems or Ada implementations may, however, require that all files to be used by a program be declared to the operating system before program execution starts, and may allow deletion of a file only after program execution has ended.

The RM allows an implementation to always raise USE_ERROR when CREATE is called in such environments. However, the RM also allows (and users would probably prefer) CREATE with mode OUT_FILE to have the same effect as OPENING an existing file in mode OUT_FILE.

To see the effect of DELETE in such environments, consider the following sequence of calls:

```
OPEN (F, OUT_FILE, "Test");  
DELETE (F); -- (1)  
OPEN (F, OUT_FILE, "Test"); -- (2)
```

If no exception is raised at (1), then the OPEN at (2) *must* raise NAME_ERROR since the file no longer exists (RM 14.3.1/13). An implementation that does not support dynamic deletion of files may, however, raise USE_ERROR at (1) (RM 14.2.1/13).

Changes from July 1982

S10. RESET raises MODE_ERROR when it attempts to change the mode of a file currently serving as a default input or default output file.

Changes from July 1980

S11. CREATE and OPEN are defined to set the page and line lengths to UNBOUNDED.

S12. The effect of OPEN and CREATE on the current column, line, and page numbers is defined.

S13. The effect of CLOSE or RESET on a file of mode OUT_FILE is defined.

S14. The effect of RESET on the line and page length is defined.

S15. The effect of RESET on the current column, line, and page numbers is defined.

Exception Conditions

The set of exceptions that can be raised is not limited to the following. Any predefined exception can be propagated by any of the subprograms defined in the input-output packages, as well as (unnameable) exceptions defined within the subprograms. All such additional exceptions must be mentioned in Appendix F for a given implementation.

CREATE Exceptions

- E1. STATUS_ERROR is raised if the internal file is already open (RM 14.2.1/4).
- E2. NAME_ERROR is raised if the NAME string does not identify an external file (RM 14.2.1/4), e.g., because the name contains an illegal character, is too long, or is ill-formed in some way.
- E3. USE_ERROR is raised if the environment does not support the creation of an external file for the specified mode with the given name and form (e.g., because the named file already exists and cannot be overwritten (RM 14.4/5)) (RM 14.2.1/4).
- E4. USE_ERROR is raised if the FORM string is not acceptable to the implementation (RM 14.2.1/4).
- E5. STORAGE_ERROR is raised if insufficient storage exists to permit creation of the data structures associated with the file.

OPEN Exceptions

- E6. STATUS_ERROR is raised if the internal file is already open (RM 14.2.1/7).
- E7. NAME_ERROR is raised if the NAME string does not identify an existing external file (RM 14.2.1/7), e.g., because the name is illegal or a file with the specified name does not exist.
- E8. USE_ERROR is raised if the environment does not support the opening of an external file for the specified mode with the given name and form (RM 14.2.1/7).

E9. **STORAGE_ERROR** is raised if insufficient storage exists to permit creation of the data structures associated with the file.

CLOSE Exceptions

E10. **STATUS_ERROR** is raised if the internal file is not open (RM 14.2.1/10).

DELETE Exceptions

E11. **STATUS_ERROR** is raised if the internal file is not open (RM 14.2.1/13).

E12. **USE_ERROR** is raised if the external file cannot be deleted (RM 14.2.1/13; see also IG 14.2.1/S).

RESET Exceptions

E13. **STATUS_ERROR** is raised if the internal file is not open (RM 14.2.1/16).

E14. **MODE_ERROR** is raised if **RESET** attempts to change the mode of a file serving as the current default input or default output file (RM 14.2.1/5).

E15. **USE_ERROR** is raised if the environment does not support resetting for the external file (RM 14.2.1/16).

MODE Exceptions

E16. **STATUS_ERROR** is raised if the internal file is not open (RM 14.2.1/19).

NAME Exceptions

E17. **STATUS_ERROR** is raised if the internal file is not open (RM 14.2.1/22).

FORM Exceptions

E18. **STATUS_ERROR** is raised if the internal file is not open (RM 14.2.1/25).

Test Objectives and Design Guidelines

T1. Check that the subprograms defined in RM 14.2.1 (**CREATE**, **OPEN**, **CLOSE**, **DELETE**, **RESET**, **MODE**, **NAME**, **FORM**, **IS_OPEN**) and the function **END_OF_FILE** are available for text files and have the correct formal parameter names. (**END_OF_FILE** is checked in IG 14.3.4/T8).

T2. Check that the following exceptions are raised:

- **STATUS_ERROR** when an internal file is open and an attempt is made to **CREATE** or **OPEN** the internal file;
- **STATUS_ERROR** when an internal file is not open and an attempt is made to **CLOSE**, **DELETE**, **RESET** a file, or determine the **MODE**, **NAME**, or **FORM** of a file;
- **NAME_ERROR** when the **NAME** string does not identify an external file for an **OPEN** or **CREATE**;
Implementation Guideline: Include a case when the external file does not already exist when an **OPEN** is attempted for an **OUT_FILE**.
- **USE_ERROR** when an implementation does not support creating or opening an external file of the specified mode;

- **USE_ERROR** when an external file cannot be **RESET** or **DELETED**.
- **MODE_ERROR** when attempting to change the mode of a file serving as the current default input or default output file.

Check whether **USE_ERROR** is raised by **CREATE** and **OPEN** when a named file already exists and **OUT_FILE** is specified as the mode.

- T3. Check that after opening (after a **CREATE** or **OPEN**) or resetting a file with mode **OUT_FILE**, the page and line length have the value zero.

Implementation Guideline: Include a case that resets a file with current mode **IN_FILE** to a new mode of **OUT_FILE**, as well as resetting an **OUT_FILE**.

Check that when resetting a file, if the old and new modes are both **OUT_FILE**, the line and page lengths are still reset to zero.

Implementation Guideline: Be sure the line and page lengths are nonzero before **RESET** is called.

- T4. Check that after opening (after a **CREATE** or **OPEN**) or resetting a file with either **MODE**, the current column, current line, and current page numbers are set to one.

Check that **RESET** does not close the file.

Check that the **MODE** parameter in **RESET** changes the mode of a given file, and if no mode is supplied, the mode is left as it was before the **RESET**.

- T5. Check that the mode **INOUT_FILE** is not allowed for **TEXT_IO**.

- T6. Check that closing or resetting an **OUT_FILE** to an **IN_FILE** has the following effect:

- a. if there is no line terminator, a line terminator, page terminator, and file terminator are written at the end of the file.
- b. if there is a line terminator but no page terminator, a page terminator and file terminator are written.
- c. if there is a page terminator, a file terminator is written.

- T7. Check that **IS_OPEN** returns the proper values:

- after the file is declared but before it is open or created;
- following **CREATE** (both successful and unsuccessful);
- after a successful **CLOSE**;
- following **OPEN** (both successful and unsuccessful);
- after a **RESET**;
- after a **DELETE**.

- T8. Check that a file may be closed and then re-opened.

Check that the name returned by **NAME** can be used in a subsequent **OPEN**.

Implementation Guideline: Close the file following the call to **NAME**, then re-open it. Otherwise the test would not be possible for an implementation that does not permit multiple internal files to be associated with an external file.

- T9. Check that **CREATE** is permitted for an **IN_FILE**.

Check whether **USE_ERROR** is raised.

- T10. Check that after a successful **DELETE** of an external file, the **NAME** of the file can be used in a **CREATE** operation.

- T11. Check whether or not more than one internal file may be associated with the same external file.
Implementation Guideline: Check both with permanent and temporary file names.
Implementation Guideline: Check by creating one file and then opening the created external file under another internal name.
- T12. Check that an external file specified by a null string NAME is not accessible after the completion of the main program, and an external file specified by a non-null string NAME is accessible after the completion of the main program.
 Check that if two files are created by specifying null names, distinct temporary files are created.
- T13. Check that the default mode for CREATE is OUT_FILE for TEXT_IO files (see IG 14.2.1/T9).
- T14. Check that an external file ceases to exist after a successful DELETE.
 Check whether or not an external file associated with more than one internal file may be DELETED.
- T15. Check that resetting a file accessed by more than one internal file does not affect the internal files that are not reset.
- T16. Determine the number of internal files an implementation can support.
- T17. Check that a null string for FORM specifies the use of the default options of the implementations, as specified in Appendix F, for the external file.
- T18. Check that FORM returns the form string for the external file.
- T19. Check that if the implementation allows alternate forms of the name or form, FORM and NAME return a full specification of the form and name.

14.3.2 Default Input and Output Files

Semantic Ramifications

S1. Note that assignment for file types is not available, since such types are limited private. Therefore, the following is illegal.

```
INPUT : FILE_TYPE := CURRENT_INPUT;    -- illegal
```

S2. Although the standard input and standard output files cannot be closed, the current default input and current default output files can be closed (and even deleted):

```
X : FILE_TYPE;
...
SET_INPUT (X);
CLOSE (X);    -- current default input file now closed
```

Subsequent attempts to read from the default input file will raise STATUS_ERROR:

```
GET (CHAR);    -- STATUS_ERROR raised
```

S3. Although the mode of a default input or output file cannot be changed by using RESET (RM 14.3.1/5), the mode can be changed by closing the file and reopening it with a different mode:

```
X : FILE_TYPE;
...
```

```

OPEN (X, OUT_FILE, "Test");
SET_OUTPUT (X);
CLOSE (X);           -- no exception
OPEN (X, IN_FILE, "Test"); -- no exception
PUT (CHAR);          -- MODE_ERROR is raised

```

The PUT raises MODE_ERROR because the default output file, X, now has mode IN_FILE.

Changes from July 1982

There are no significant changes.

Changes from July 1980

S4. SET_INPUT and SET_OUTPUT raise MODE_ERROR when the mode of their arguments is not correct.

S5. STATUS_ERROR cannot be raised by CURRENT_INPUT and CURRENT_OUTPUT.

Exception Conditions

SET_INPUT Exceptions

- E1. STATUS_ERROR is raised if the given file is not open.
- E2. MODE_ERROR is raised if the mode of the given file is not IN_FILE.

SET_OUTPUT Exceptions

- E3. STATUS_ERROR is raised if the given file is not open.
- E4. MODE_ERROR is raised if the mode of the given file is not OUT_FILE.

Test Objectives and Design Guidelines

- T1. Check that the standard input and output files exist and are initially open.
- T2. Check that CURRENT_INPUT and CURRENT_OUTPUT initially correspond to the standard files.

Implementation Guideline: Files cannot be compared for equality, since they have a limited private type. It should suffice to compare NAME(STANDARD_INPUT) with NAME (CURRENT_INPUT).

- T3. Check that SET_INPUT and SET_OUTPUT can be used.

Check that calls to SET_INPUT or SET_OUTPUT do not redefine or close the corresponding standard files.

Check that the formal parameter name is FILE.

- T4. Check that after the default files have been redefined, input and output on the standard files is still properly handled.

Implementation Guideline: Use STANDARD_INPUT and STANDARD_OUTPUT to denote the files.

- T5. Check that the standard input and output files cannot be opened, closed, reset, or deleted.
- T6. Check that STATUS_ERROR is raised when SET_INPUT and SET_OUTPUT are called with a closed file.
- T7. Check that MODE_ERROR is raised if the parameter to SET_INPUT has mode OUT_FILE or the parameter to SET_OUTPUT has mode IN_FILE.
- T8. Check that the file used as the parameter to SET_INPUT or SET_OUTPUT can be closed, causing the corresponding default file to become closed.

Check that when the file serving as a default file is closed and reopened with a changed mode, the mode of the default file is also changed.

14.3.3 Specification of Line and Page Lengths

Semantic Ramifications

- S1. It is not permissible to raise `USE_ERROR` if, at the time of a call to `SET_LINE_LENGTH`, the current column exceeds the value given in the `TO` parameter. The next `PUT` operation will output a `NEW_LINE` (e.g., see RM 14.3.6/6).
- S2. A similar consideration applies to `SET_PAGE_LENGTH`. After the call, the current page will be terminated the next time a `NEW_LINE` operation is performed.
- S3. If an implementation requires a bounded line length for an external file, `SET_LINE_LENGTH` (`LINE_LENGTH`) can raise `USE_ERROR`, since `LINE_LENGTH` is zero after a file is opened, created, or reset to `OUT_FILE` (see IG 14.3.1/S).

Changes from July 1982

- S4. There are no significant changes.

Changes from July 1980

- S5. `SET_PAGE_LENGTH` and `PAGE_LENGTH` are added.
- S6. The `TO` parameter subtype does not include negative values, and is a value of type `COUNT`, not `INTEGER`.

Exception Conditions

`SET_LINE_LENGTH` Exceptions

- E1. `STATUS_ERROR` is raised if the file is not open.
- E2. `MODE_ERROR` is raised if the file's mode is `IN_FILE`.
- E3. `USE_ERROR` is raised if the specified line length is inappropriate for the external file.
- E4. `CONSTRAINT_ERROR` is raised if the value of `TO` is negative or greater than `COUNT'LAST`.

`SET_PAGE_LENGTH` Exceptions

- E5. `STATUS_ERROR` is raised if the file is not open.
- E6. `MODE_ERROR` is raised if the file's mode is `IN_FILE`.
- E7. `USE_ERROR` is raised if the specified page length is inappropriate for the external file.
- E8. `CONSTRAINT_ERROR` is raised if the value of `TO` is negative or greater than `COUNT'LAST`.

`LINE_LENGTH` Exceptions

- E9. `STATUS_ERROR` is raised if the file is not open.
- E10. `MODE_ERROR` is raised if the file's mode is `IN_FILE`.

PAGE_LENGTH Exceptions

E11. STATUS_ERROR is raised if the file is not open.

E12. MODE_ERROR is raised if the file's mode is IN_FILE.

Test Objectives and Design Guidelines

T1. Check that LINE_LENGTH and PAGE_LENGTH have the value zero when a file is created, opened, or reset, and the mode is OUT_FILE (see IG 14.3.1/T3).

Check that when the line and page length are nonzero, line and page terminators are output at the appropriate points.

Check that SET_LINE_LENGTH and SET_PAGE_LENGTH take parameters of type COUNT (not INTEGER), and similarly, that LINE_LENGTH and PAGE_LENGTH return values of type COUNT.

Check that when the file parameter is omitted, the current default output file is used.

T2. Check that SET_LINE_LENGTH, SET_PAGE_LENGTH, LINE_LENGTH, and PAGE_LENGTH raise MODE_ERROR when called with a file of mode IN_FILE.

T3. Check that SET_LINE_LENGTH, SET_PAGE_LENGTH, LINE_LENGTH, and PAGE_LENGTH raise STATUS_ERROR when applied to a closed file.

T4. Check that SET_LINE_LENGTH and SET_PAGE_LENGTH raise USE_ERROR when the specified lengths are inappropriate.

T5. Check that the line and page length can be set to a value shorter than the current length of the line.

Check that the line and page length can be altered dynamically several times and reset to zero; check that the zero value is taken to indicate that the length is unbounded.

T6. Check that CONSTRAINT_ERROR is raised if the value of TO is negative or greater than COUNT'LAST when COUNT'LAST is less than COUNT'BASE'LAST.

14.3.4 Operations on Columns, Lines, and Pages**Semantic Ramifications**

S1. Column number, line number, and page number are properties of the internal file, not of the external file. In particular, if an implementation allows the input and output files to correspond to the same external file (as might be the case for an interactive terminal), those quantities will not always have the same value.

S2. Note that NEW_LINE always outputs a line terminator before checking to see if the page length has been exceeded.

S3. Although the RM states that for SET_COL, SET_LINE, COL, LINE, and PAGE the default is the current default output file, this does not preclude performing these operations on the current default input file. Specifically,

```
SET_COL (CURRENT_INPUT, 10);
```

is perfectly legal.

S4. An implementation may choose to raise USE_ERROR when SKIP_LINE with SPACING > 1 is attempted on an interactive input device, since USE_ERROR can be raised "if an operation

is attempted that is not possible for reasons that depend on characteristics of the external file" (RM 14.4/5).

S5. Because SET_COL for an IN_FILE performs GET operations, the column, line, and page numbers are updated after each GET. In particular, if END_ERROR is raised, the page number must reflect the number of page terminators actually read. (Note that after END_ERROR is raised, the column and line numbers will both be one, since a page terminator is the last entity read before END_ERROR is raised.)

S6. The phrase "has the effect of calling NEW_LINE," used in defining SET_COL and SET_LINE for an OUT_FILE, and similarly, the phrase "has the effect of calling SKIP_LINE" — used in the definition of the same operations for an IN_FILE — imply the proper treatment of the end-of-page condition.

S7. The semantics of CLOSE (RM 14.3.1/3) and RESET (RM 14.3.1/4) ensure that every text file is logically terminated with the sequence: line terminator, page terminator, file terminator. When reading, it is not possible to skip a line terminator at the end of a page without also skipping the page terminator (see RM 14.3.4/8). Hence, it is not possible to call END_OF_FILE when positioned after a line terminator and before the sequence, page terminator, file terminator. Hence, RM 14.3.4/24 covers all possible cases.

S8. Note that on input, SET_COL reads until it finds a line having a character at the specified position. A line terminator is not a character. Now consider a file created by the following calls:

```
PUT ("ABC");
NEW_LINE;
PUT ("DEFG");
NEW_LINE;
```

Assume that the file is RESET for input and that the command SET_COL(4) is issued followed by GET(CHAR). The character read will be 'G' even though COL equals four after character C is skipped. This shows that it is incorrect to implement SET_COL by reading until COL equals the specified value — one must read until COL equals the specified value *and* END_OF_LINE is false.

S9. Note that checking for END_OF_LINE requires reading one character to see if it is a line terminator (or the beginning of a line terminator sequence). If END_OF_LINE is false, then the next call to GET should return the character END_OF_LINE looked at, i.e., at least one character look-ahead is needed to support the END_OF_LINE operation. Depending on how END_OF_PAGE is supported, more than one character of look-ahead may be needed to implement END_OF_PAGE.

S10. Note that if the file is positioned before the final page terminator, END_OF_FILE will be true, but SKIP_PAGE will not raise END_ERROR, nor will SKIP_LINE. It is incorrect to assume that END_OF_FILE being TRUE implies that all subsequent read operations will raise END_ERROR.

Changes from July 1982

S11. There are no significant changes.

Changes from July 1980

S12. The default file for COL and LINE is specified as the current default output file.

S13. The semantics of SET_COL are defined more precisely for both IN_FILES and OUT_FILES.

S14. The semantics of SET_LINE_LENGTH are clarified.

S15. New procedures and semantics are added to account for the existence of page terminators: NEW_PAGE, SKIP_PAGE, END_OF_PAGE, END_OF_FILE, SET_LINE, PAGE.

Exception Conditions

E1. STATUS_ERROR is raised for each operation when applied to an unopen file.

NEW_LINE Exceptions

E2. MODE_ERROR is raised if the current mode is not OUT_FILE.

E3. CONSTRAINT_ERROR is raised if the SPACING is less than one or greater than COUNT'LAST.

SKIP_LINE Exceptions

E4. MODE_ERROR is raised if the current mode is not IN_FILE.

E5. END_ERROR is raised if the number of lines to be skipped exceeds the number of lines remaining in the file.

E6. CONSTRAINT_ERROR is raised if the SPACING is less than one or greater than COUNT'LAST.

END_OF_LINE Exceptions

E7. MODE_ERROR is raised if the current mode is not IN_FILE.

NEW_PAGE Exceptions

E8. MODE_ERROR is raised if the current mode is not OUT_FILE.

SKIP_PAGE Exceptions

E9. MODE_ERROR is raised if the current mode is not IN_FILE.

E10. END_ERROR is raised if the number of pages to be skipped exceeds the number of pages remaining in the file.

END_OF_PAGE Exceptions

E11. MODE_ERROR is raised if the current mode is not IN_FILE.

END_OF_FILE Exceptions

E12. MODE_ERROR is raised if the mode is not IN_FILE.

SET_COL Exceptions

E13. LAYOUT_ERROR is raised if the mode is OUT_FILE, if the value specified by TO exceeds LINE_LENGTH, and if LINE_LENGTH is greater than zero.

E14. END_ERROR is raised if the mode is IN_FILE and the longest line remaining to be read has fewer than the required number of characters.

E15. CONSTRAINT_ERROR is raised if the value of TO is less than one or greater than COUNT'LAST.

SET_LINE Exceptions

- E16. **LAYOUT_ERROR** is raised if the mode is **OUT_FILE**, if the value specified by **TO** exceeds **PAGE_LENGTH**, and if **PAGE_LENGTH** is greater than zero.
- E17. **END_ERROR** is raised when the mode is **IN_FILE** and an attempt is made to set the line number past a file terminator.
- E18. **CONSTRAINT_ERROR** is raised if the value specified by **TO** is less than one.

COL Exceptions

- E19. **LAYOUT_ERROR** is raised if the value of the current column number exceeds **COUNT'LAST**.

LINE Exceptions

- E20. **LAYOUT_ERROR** is raised if the value of the current line number exceeds **COUNT'LAST**.

PAGE Exceptions

- E21. **LAYOUT_ERROR** is raised if the value of the current page number exceeds **COUNT'LAST**.

Test Objectives and Design Guidelines

- T1. Check that the formal parameters of each operation are named correctly.
- T2. Check that **NEW_LINE**:
- raises **MODE_ERROR** when the file mode is **IN_FILE**.
 - has an optional **SPACING** parameter with default value one.
 - operates on the current default output file if no file is specified.
 - when the page length is nonzero and the current line number equals (or exceeds) the maximum page length, **NEW_LINE** outputs a line terminator followed by a page terminator, increments the current page number and sets the current line number to one; check that, if necessary, several page terminators are output and the page number is incremented by a value greater than one. Check that empty pages have the appropriate number of line terminators (i.e., the number specified by the page length).
Implementation Guideline: Try one case where the current line number exceeds the page length.
 - outputs **SPACING** line terminators when **SPACING** is greater than one.
 - sets the current column number to one.
 - raises **CONSTRAINT_ERROR** if **SPACING** is zero or negative or greater than **COUNT'LAST** when **COUNT'LAST** < **COUNT'BASE'LAST**.
- T3. Check that **SKIP_LINE**:
- raises **MODE_ERROR** when the file mode is **OUT_FILE**.
 - skips a single line terminator if the **SPACING** parameter is omitted.
 - operates on the current default input file if the file parameter is omitted.

- is performed SPACING times for a SPACING greater than one.
- sets the current column number to one.
- raises CONSTRAINT_ERROR if SPACING is zero or negative or greater than COUNT'LAST when COUNT'LAST < COUNT'BASE'LAST.
- increments the current line number by one and sets the current column number to one if the line terminator is not followed by a page terminator.
- sets the current line and column numbers to one and increments the current page number by one when the line terminator is followed by a page terminator.
- raises END_ERROR if an attempt is made to skip a file terminator.

Implementation Guideline: Check that the file terminator is not skipped. Check that the column number is correct.

T4. Check that END_OF_LINE:

- raises MODE_ERROR when applied to an OUT_FILE.
- operates on the current default input file if no file is specified.
- returns the correct value when positioned at the beginning and the end of a line, and when positioned just before the file terminator.

T5. Check that NEW_PAGE:

- outputs a line terminator followed by a page terminator if the current line is not at column 1 or if the current page is at line 1; if the current line is at column 1, outputs a page terminator only.
- operates on the current default output file if no file is specified.
- raises MODE_ERROR if the file specified has mode IN_FILE.
- increments the current page number and sets the current column and line numbers to one.

T6. Check that SKIP_PAGE:

- reads and discards characters and line terminators until a page terminator is read.
- adds one to the current page number and sets the current column and line numbers to one.
- raises MODE_ERROR when the mode of the specified file is not IN_FILE.
- raises END_ERROR when the file is positioned before the file terminator but not when the file is positioned before the final page terminator.
- operates on the current default input file when no file is specified.

T7. Check that END_OF_PAGE:

- returns TRUE if positioned just before a file terminator or at the end of a page (before a line and page terminator) and otherwise returns FALSE.
- returns the same value when called more than once at the same file position (i.e., check that END_OF_PAGE does not advance the file position).

- raises `MODE_ERROR` if the specified file is not of mode `IN_FILE`.
- operates on the current default input file when no file is specified.

T8. Check that `END_OF_FILE`:

- returns `TRUE` if positioned before the last page terminator of the file or before a file terminator, and `FALSE` otherwise.
- returns the same value when called more than once at the same file position.
Implementation Guideline: Check that `END_ERROR` is not raised when `END_OF_FILE` is called more than once when the file is positioned before the file terminator.
- raises `MODE_ERROR` if the mode of the specified file is not `IN_FILE`.
- operates on the current default input file if no file is specified.

T9. Check that `SET_COL`:

- raises `LAYOUT_ERROR` if the line length is bounded and the given column position exceeds the line length for files of mode `OUT_FILE`.
- raises `CONSTRAINT_ERROR` if the given column position is zero or negative (even when the file is closed) or greater than `COUNT'LAST` when `COUNT'LAST < COUNT'BASE'LAST`.
- for `OUT_FILES`, if the specified column number is:
 - greater than the current column number, outputs spaces until the specified and actual column numbers are equal;
 - less than the current column number, outputs a line terminator and page terminator if necessary, then outputs the required number of spaces.
- for `IN_FILES`, reads zero or more characters until the specified column number equals the current column number and `END_OF_LINE` is false.
- raises `END_ERROR` for files of mode `IN_FILE` when an attempt is made to set the column past a file terminator.
- operates on the current default output file if no file is specified.

T10. Check that `SET_LINE`:

- raises `LAYOUT_ERROR` if the page length is bounded and the given line number exceeds the page length.
- raises `CONSTRAINT_ERROR` if the given line number is zero or negative (even if the file is closed) or greater than `COUNT'LAST` when `COUNT'LAST < COUNT'BASE'LAST`.
- for `OUT_FILES`, if the specified line number is:
 - greater than the current line number, outputs line terminators until the current line number equals the specified number;
Implementation Guideline: Try one case where the specified number equals the page length.
 - less than the current line number, outputs a page terminator and then the required number of line terminators.

- for IN_FILES, skips zero or more lines until the current line number equals the specified line number.
- raises END_ERROR for files of mode IN_FILE when an attempt is made to set the current line past a file terminator.
- operates on the current default output file if no file is specified.

T11. Check that COL:

- returns the value of the current column number.
- raises LAYOUT_ERROR when the value of the column number exceeds COUNT'LAST.
- operates on the current output file if no file is specified, and operates on files of both IN_FILE and OUT_FILE.

T12. Check that LINE:

- returns the value of the current line number.
- raises LAYOUT_ERROR when the value of the line number exceeds COUNT'LAST.
- operates on the current output file if no file is specified, and operates on files of both IN_FILE and OUT_FILE.

T13. Check that PAGE:

- returns the value of the current page number.
- raises LAYOUT_ERROR when the value returned exceeds COUNT'LAST.
- operates on the current output file if no file is specified, and operates on files of both IN_FILE and OUT_FILE.

T14. Check that all operations raise STATUS_ERROR when applied to closed files.

14.3.5 GET and PUT Procedures

Semantic Ramifications

S1. Since the treatment of control characters by TEXT_IO is implementation-dependent (RM 14.3/7), the compiler and the run-time package need not be consistent in the following aspects:

- The compiler must consider one or more carriage return, line feed, vertical tabulate, or form feed, as causing passage to a new line (RM 2.2/3). The number of line marks signified by a sequence of these characters is implementation-dependent for the compiler. TEXT_IO may (but need not) follow the same rule by considering equivalent sequences to form line marks.
- When processing program text, the compiler must disallow backspace, delete, null, and other nongraphic characters other than format effectors, (RM 2.1/1) and treat horizontal tabulate as a legal separator (RM 2.2/2). TEXT_IO must also accept horizontal tabulate when reading numeric input; the treatment of the other nongraphic characters is implementation-dependent.

S2. The first sentence of RM 14.3.5/10 says END_ERROR is raised by a GET procedure "if an

attempt is made to skip a file terminator." Such attempts occur while skipping line terminators, page terminators, and blanks that precede enumeration or numeric literals (RM 14.3.5/6). Once reading of a lexical element has begun, the line and page terminator that precede a file terminator (RM 14.3.1/3, /4) will end the lexical element; hence, `END_ERROR` will not be raised. It should be noted that `END_ERROR` can also be raised when getting a numeric or an enumeration value from a string (RM 14.3.7/14, RM14.3.8/18, and RM 14.3.9/11) if the string is null or consists only of spaces and horizontal tabulation characters. Finally, `END_ERROR` can be raised by `GET_LINE` (RM 14.3.6/15).

s3. When `GET` or `GET_LINE` is applied to a file associated with an input device, an implementation may choose to assume that a file terminator can never be input, or it may treat a particular control character (or a sequence of control characters) as signifying a file terminator. Whether `END_ERROR` can ever be raised is thus implementation-dependent.

Changes from July 1982

s4. See IG 14.3.6-9/Changes.

Changes from July 1980

s5. See IG 14.3.6-9/Changes.

Exception Conditions

The exceptions raised for each operation are specified in subsequent sections.

E1. `DEVICE_ERROR` may be raised by any operation due to a system malfunction.

Test Objectives and Design Guidelines

Tests for these functions are specified in the sections discussing `TEXT_IO` for individual data types.

14.3.6 Input-Output of Characters and Strings

Semantic Ramifications

s1. `GET` for strings and characters does not require that the input satisfy the lexical rules for string and character literals (unlike `GET` for numeric and enumeration types). Instead, `GET` inputs a sequence of ASCII characters, exclusive of any (control) characters that signify line, page, or file terminators. The treatment of ASCII control characters is implementation-dependent (RM 14.3/7). In particular, since the horizontal tabulation character (HT) is a control character, an attempt to read or write it using the `GET` and `PUT` procedures defined in RM 14.3.6 has an implementation-defined effect. Since RM 14.3.5/5-6, however, defines an implementation-independent effect of reading HT when inputting scalar values, most implementations will probably define `GET` and `PUT` for characters and strings to accept HT as a value. It is likely that only characters (and character sequences) used to implement line, page, and file terminators will be ignored by `GET`.

s2. Similarly, `PUT` for strings simply outputs ASCII character codes, although the treatment of characters used to signify line, page, and file terminators is implementation-dependent. In particular, the line and page count need not be affected.

s3. Note that by the definition of `GET` for strings, no translation is performed on the input. The transliteration rules of RM 2.10 do not apply.

s4. Note that when `GETting` a string, line and page terminators are skipped over until the

required number of nonterminator characters has been read. If fewer than the required number are left to be read, then `END_ERROR` will be raised.

S5. Note that `GET` and `PUT` operations for characters and strings can only be applied to a file, unlike integer, real, and enumeration type operations, which can be applied to strings as well.

S6. Note that the `GET` and `PUT` operations for character and string do not have a `WIDTH` parameter.

S7. The usual semantics of out parameters and exceptions imply that if `END_ERROR` is raised by `GET_LINE`, the value of the `LAST` parameter must be unchanged from its value at the time of call (see RM 6.2/6). Of course, whether the value of the `ITEM` parameter is affected is not defined by the language, since `ITEM` has the type `STRING` in this case and so is not necessarily passed by copy. If `END_ERROR` is raised by `GET` for characters, the value of the `ITEM` parameter is unchanged since `ITEM` is passed by copy in this case.

S8. RM 14.3.6/13 says that for `GET_LINE`:

Reading stops if the end of the line is met, in which case the procedure `SKIP_LINE` is then called (in effect) with a spacing of one; reading also stops if the end of the string is met.

This sentence specifies two conditions that are not mutually exclusive. In particular, if the end of the string and the end of the line occur simultaneously (i.e., if the number of characters remaining to be read on the line is the same as the length of the string), then the line terminator is skipped. Hence, when `GET_LINE` is called at the end of a line with a null string parameter, the effect is equivalent to `SKIP_LINE`, and the value of `LAST` is one less than the lower bound of the `ITEM` parameter.

S9. After skipping the last page terminator in a file, only the file terminator remains. If `GET_LINE` is called, `END_OF_LINE` is true (RM 14.3.4/12), so no characters are read (the end of the line has been met) and `SKIP_LINE` is called. The `SKIP_LINE` call attempts to skip the file terminator, and therefore raises `END_ERROR`.

Changes from July 1982

S10. `GET_LINE` no longer skips leading line terminators.

S11. `LAST` is the index of the last character replaced, not the number of characters read.

Changes from July 1980

S12. The `GET_STRING` subprogram no longer exists.

S13. The `GET_LINE` subprogram is redefined as a procedure.

Exception Conditions

GET Exceptions

- E1. `STATUS_ERROR` is raised if the file is not open.
- E2. `MODE_ERROR` is raised if the file's mode is not `IN_FILE`.
- E3. `END_ERROR` is raised by `GET` for characters if only line and page terminators remain to be read.
- E4. `END_ERROR` is raised by `GET` for strings if the number of characters remaining in a file, exclusive of any characters used to signify line and page terminators, is less than the length of the `ITEM` parameter.

GET_LINE Exceptions

- E5. STATUS_ERROR is raised if the file is not open.
- E6. MODE_ERROR is raised if the file's mode is not IN_FILE.
- E7. END_ERROR is raised for GET_LINE if and only if no characters, line terminators, or page terminators remain to be read.

PUT Exceptions

- E8. STATUS_ERROR is raised if the file is not open.
- E9. MODE_ERROR is raised if the file's mode is not OUT_FILE.

PUT_LINE Exceptions

- E10. STATUS_ERROR is raised if the file is not open.
- E11. MODE_ERROR is raised if the file's mode is not OUT_FILE.

Test Objectives and Design Guidelines

- T1. Check that GET and PUT for characters and strings, and GET_LINE and PUT_LINE for lines, raise STATUS_ERROR if the file is not open.

Check the names of the formal parameters for each of the subprograms.

- T2. Check that GET for characters and strings:

- reads ASCII graphic characters (see T7 for control characters).
- allows a STRING to span over more than one line, skipping intervening line and page terminators.
- accepts a null string actual parameter and a string slice.
- properly sets the line, page, and column numbers after the operation.
- raises MODE_ERROR for (open) files of mode OUT_FILE.
- can operate on any file of mode IN_FILE, and if no file is specified, the current default input file is used.

- T3. Check that the last graphic character in a file may be read without raising END_ERROR, and that after the last character of the file has been read, any attempt to read further characters will raise END_ERROR. Check that END_ERROR is raised by:

- GET for characters if only line and page terminators remain in the file.
- GET for STRING if the graphic characters remaining in the file are fewer than required to fill the string.
- GET_LINE if and only if no characters, line terminators, or page terminators remain to be read (whether or not the ITEM parameter is a null string).

Implementation Guideline: The file must be positioned before the file terminator.

- T4. Check that GET_LINE:

- may be called to return an entire line.
- may be called to return the remainder of a partly read line.

- if called when the input is at the end of a line will effectively SKIP_LINE and return no characters, even when the string parameter is a null string.
- if called with a string exactly equal to the (nonzero) number of characters remaining on a line, skips the line terminator after reading all the characters.
- returns in the parameter LAST the index value of the last character read. Verify that LAST is one less than ITEM's lower bound when no characters are read.

T5. Check that PUT for character and string parameters:

- does not update the line number if the line length is unbounded, only the column number.
- raises MODE_ERROR for (open) files of mode IN_FILE.
- outputs a line terminator (and possibly a page terminator) after outputting N characters, if the line length, L, is bounded ($L \neq 0$), C is the column number at the time PUT is called, and N is the smallest non-negative number such that $N + C > L$; thereafter, outputs a line terminator after outputting L characters and before outputting the next character.
Implementation Guideline: Be sure no line terminator is output if the string is null or if the last character output by a call to PUT was in column L.
Implementation Guideline: If the current column number exceeds L when PUT is called, a line terminator is output before any characters are output.
- can operate on any file of mode OUT_FILE, and if no file is specified the current default output file is used.

T6. Check that PUT_LINE:

- will accept a null string and output, correspondingly, a line terminator.
- will output the given string on more than one line when the line length is bounded and the number of characters to be output is greater than the line length.
- can be used only for STRING parameters.

T7. Check the effect of reading and writing control characters. In particular, check to see if:

- reading or writing an HT increments the column count by one and inputs or outputs an HT character (an increment greater than one is possible if the implementation simulates movement to a tab stop).
- all control characters can be written and read, and their effect on line count, page count, and file termination.
Implementation Guideline: Keep in mind that ASCII NUL or EOT may signify end of file or be illegal to write.

14.3.7 Input-Output for Integer Types

Semantic Ramifications

- S1. Note that a *blank* is defined as a space or a horizontal tabulation character (RM 14.3.5/5). Hence, leading spaces or tabs are skipped when reading an integer literal.
- S2. When reading with a WIDTH of zero, reading stops when the initial sequence of the syntax

of an integer literal is no longer satisfied (RM 14.3.5/5, /6). RM 2.4/1 defines an integer literal as a numeric literal without a decimal point. Hence, the *syntax* of an integer literal permits a minus sign in its exponent, although RM 2.4.1/4 imposes an additional restriction that affects the raising of DATA_ERROR. Hence, when reading the following input:

```
14.3X      -- next character read is '.'
14EX       -- next character read is 'X'
14E-3X     -- next character read is 'X'
```

DATA_ERROR is raised by GET and the next character to be read is indicated by the comments. Since lexical elements cannot extend across line boundaries (RM 2.2/1, /2), and since GET analyzes input sequences of characters as lexical elements (RM 14.3/2), the reading of a numeric literal stops when a line terminator is encountered.

S3. The syntax of a based_integer (see RM 2.4.2/2) allows any letter as an extended_digit. Therefore, if the input file contains, for example,

```
16#FGH#
```

then all characters up to the second # must be read before DATA_ERROR is raised. It would be incorrect to leave the input positioned at the letter G. Similarly, if the input contains:

```
10#ABC#
```

then the letters must be read.

S4. The letter E in the exponent part, and the letters used as extended digits in based notation, may appear in upper or lower case on input (RM 2.4.1/3). No exponent is provided on output.

S5. Numeric literals input in based notation may have the # character replaced uniformly (within a literal) by : characters (RM 2.10/3), since the usual rules for analyzing lexical elements are used (RM 14.3/2). The implementation has the same freedom for output. In particular, if the compiler runs in an environment supporting the # character, but the object code runs in an environment supporting only the colon, different forms of literals will be used for input-output vs. source code.

S6. When WIDTH is nonzero, exactly WIDTH characters are read (unless a line terminator is seen first; then only characters preceding the line terminator are read):

```
143e3XYZ
```

Calling GET with a WIDTH of 6 causes 6 characters to be read. DATA_ERROR is raised since "143e3X" does not satisfy the syntax of a numeric literal. Note that DATA_ERROR would also be raised if X were a space character. DATA_ERROR will not be raised if X is replaced with a line terminator, in which case, only 5 characters will be read.

S7. Note that when WIDTH is nonzero, reading stops when a line terminator is encountered. If the file is positioned just before the file terminator, END_OF_LINE is true, but an attempt to read with a nonzero WIDTH will cause DATA_ERROR to be raised (RM 14.3.5/10), since a file terminator is not a line terminator and the attempt to read will encounter the file terminator without first encountering a line terminator.

S8. Note that an attempt to read too large a value raises DATA_ERROR or CONSTRAINT_ERROR, but not NUMERIC_ERROR:

```
subtype TWO is INTEGER range 0 .. 2;
package INT_IO is new INTEGER_IO(TWO);
ITEM : TWO RANGE 0 .. 1;
BIG  : INTEGER;
```

```

LAST : POSITIVE;
...
INT_IO.GET ("3", ITEM, LAST);           -- DATA_ERROR
INT_IO.GET ("3", BIG, LAST);           -- DATA_ERROR
INT_IO.GET ("1E10_000_000_000_000", ITEM, LAST); -- DATA_ERROR
INT_IO.GET ("2", ITEM, LAST);          -- CONSTRAINT_ERROR

```

The first three calls raise `DATA_ERROR` because the values being input exceed the range of the subtype used to instantiate `INT_IO`. Note that in the second case, `DATA_ERROR` is raised even though 3 belongs to `BIG`'s subtype. The last call raises `CONSTRAINT_ERROR` because 2 in `TWO` is `TRUE`, i.e., 2 belongs to the subtype used to instantiate `INTEGER_IO`, but 2 exceeds 1, the upper bound for `ITEM`, which is the actual parameter used in the `GET` call. In short, `DATA_ERROR` is raised if the input value lies outside the range of the subtype used in an instantiation. `CONSTRAINT_ERROR` is raised if the input value lies inside the range of the subtype used in the instantiation, but outside the range of the actual parameter's subtype.

S9. The minimum length needed to output an integer number to a file depends on the value being output and on the base of the output. Note that a sign position is needed only for negative numbers. The minimum number of digits to be output depends on the absolute magnitude of `ITEM`'s value. The minimum number is 1 if `ITEM` is zero. Otherwise, the minimum number of digits is:

$$\text{floor}(\log_b(\text{abs}(\text{ITEM}))) + 1,$$

where \log_b is the logarithm to base b , and the *floor* function yields the largest integer value less than or equal to its argument. Hence, $\text{floor}(1.9) = 1$, and $\text{floor}(2.0) = 2$. Some values yielded by this expression are:

ITEM	b = 10	b = 2
10	2	4
7	1	3
-9	1	4

The expression `BOOLEAN'POS(ITEM < 0)` yields 1 if `ITEM` is negative; otherwise, zero. If the base is not 10, then the value of the base plus two # characters are output to form a based literal. Hence, the minimum field size needed to represent a nonzero integer value is given by the following expression when base is 10:

```

BOOLEAN'POS(ITEM < 0)           -- minus sign if necessary
+ floor(log10(abs(ITEM)))+1)    -- number of digits

```

When the base is not 10, the minimum field size is:

```

BOOLEAN'POS(ITEM < 0)           -- minus sign if necessary
+ 1 + BOOLEAN'POS(base > 9)    -- 1 or 2 digits for base
+ 1                             -- the first # character
+ floor(logb(abs(ITEM)))+1)    -- number of digits
+ 1                             -- the last # character

```

Hence, the smallest field size for a base 10 integer is one; the smallest field for a number output in bases 2 through 9 is four; and the smallest field for a number in bases 11 through 16 is five. All these minimums are increased by one when the value being output is negative.

Changes from July 1982

S10. `PUT` for a based literal outputs letters in upper case.

AD-A189 647

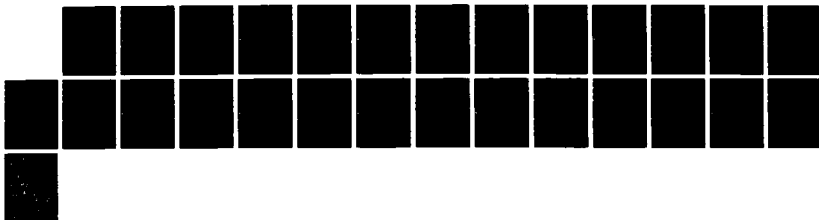
THE ADA (TRADE NAME) COMPILER VALIDATION CAPABILITY
IMPLEMENTERS' GUIDE VERSION 1(U) SOFTECH INC WALTHAM MA
J B GOODENOUGH DEC 86

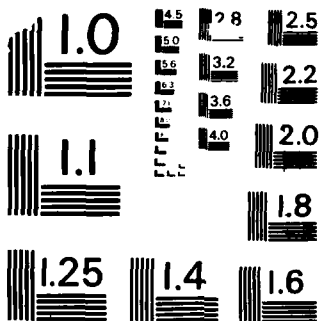
9/9

UNCLASSIFIED

F/G 12/5

NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

S11. When WIDTH is nonzero, leading line terminators are not skipped. Instead reading stops when a line terminator is seen or when WIDTH characters have been read, whichever comes first. This means that reading does not necessarily stop when the syntax of a numeric literal is violated.

S12. GET from a STRING does not replace the value of LAST with an "undefined" value when an exception is raised. Instead, the normal semantics of out parameters defines the effect on LAST, i.e., the value of LAST is unchanged (since LAST is a scalar parameter). The July 1982 wording implied that subsequent accesses to LAST could be erroneous (see RM 3.2.1/18).

Changes from July 1980

S13. GET has an optional WIDTH parameter.

S14. Default parameter values are defined by variables declared in the generic package, and hence, these defaults can be changed dynamically during program execution.

S15. GET and PUT are defined for STRINGs as well as files.

S16. DATA_ERROR is raised by GET (instead of CONSTRAINT_ERROR) if the value read is not in the range of the subtype used to instantiate the package.

Exception Conditions

GET Exceptions (file input)

E1. STATUS_ERROR is raised if the file is not open.

E2. MODE_ERROR is raised if the file's mode is not IN_FILE.

E3. DATA_ERROR is raised if:

- the input does not have the syntax of a numeric literal, has a decimal point, or has a negative exponent.
- the number being input lies outside the range specified for the integer subtype used in the instantiation of INTEGER_IO.

E4. CONSTRAINT_ERROR is raised if:

- the WIDTH parameter has a negative value or exceeds FIELD'LAST (and FIELD'LAST < INTEGER'LAST).
- the value read does not satisfy the actual parameter's range and the actual parameter's range is a subset of the range of the subtype used to instantiate INTEGER_IO.

E5. END_ERROR is raised if:

- WIDTH is zero and the only characters read are spaces, horizontal tabulation characters, line terminators, and page terminators or,
- WIDTH is nonzero, and the file is positioned just before the file terminator.

GET Exceptions (string input)

E6. DATA_ERROR is raised if:

- the input does not have the syntax of a numeric literal, has a decimal point, or has a negative exponent.

- the number being input lies outside the range specified for the integer subtype used in the instantiation of `INTEGER_IO`.

- E7. `CONSTRAINT_ERROR` is raised if the value read does not satisfy the actual parameter's range and the actual parameter's range is a subset of the range of the subtype used to instantiate `INTEGER_IO`.
- E8. `END_ERROR` is raised if the only characters read are spaces, horizontal tabulation characters, line terminators, and page terminators or if the input string is null.

PUT Exceptions (file output)

- E9. `STATUS_ERROR` is raised if the file is not open.
- E10. `MODE_ERROR` is raised if the file's mode is not `IN_FILE`.
- E11. `LAYOUT_ERROR` is raised (RM 14.3.5/10) if the line length (LL) for the file is fixed (i.e., not zero), and:
- $LL < \text{the minimum required length (whether or not WIDTH is zero), or}$
 - $WIDTH > LL$.
- E12. `CONSTRAINT_ERROR` is raised
- if the value of `ITEM` lies outside the range of the subtype used to instantiate `INTEGER_IO`.
 - when the value specified for `WIDTH` is negative or exceeds `FIELD'LAST`, and $FIELD'LAST < INTEGER'LAST$.
 - when the value specified for `BASE` is not in the range 2 through 16.

PUT Exceptions (string output)

- E13. `LAYOUT_ERROR` is raised if the length of the string variable is less than the minimum required length for the value being output (RM 14.3.5/10).
- E14. `CONSTRAINT_ERROR` is raised:
- if the value of `ITEM` lies outside the range of the subtype used to instantiate `INTEGER_IO`.
 - when the value specified for `BASE` is not in the range 2 through 16.

Test Objectives and Design Guidelines

- T1. Check that each operation raises `STATUS_ERROR` if the file is not open.
- T2. Check that `INTEGER_IO` can be instantiated for user defined types.
- T3. Check that `PUT` and `GET` for the predefined type `INTEGER` are not available without instantiation of `INTEGER_IO`.
- T4. Check that `GET` from a file:
- correctly reads nonbased literals without an exponent when the value read is the minimum or maximum value for an integer type.
Implementation Guideline: Use each of the predefined integer types.
 - correctly reads a nonbased integer literal containing an exponent, whether or

not the exponent is given in upper case, and whether or not a plus sign is explicitly provided in the exponent.

- correctly reads a based literal without an exponent, for bases 2 through 16, when the value read is the minimum or maximum value for an integer type; check that any letters used may be in upper or lower case.

Implementation Guideline: Use each of the predefined integer types.

- correctly reads a based literal containing an exponent, for bases 2 through 16, independently of the case of the exponent or the letters used in the literal, and whether or not a plus sign is explicitly provided in the exponent.
- allows : instead of # in based literals.
- uses the default input file if no file is explicitly specified.
- skips leading spaces, horizontal tabulation characters, line terminators, and page terminators if WIDTH is zero.
- when WIDTH is nonzero, reads at most WIDTH characters or up to the next line terminator; counts leading spaces and horizontal tabulation characters as part of the WIDTH.
- when WIDTH is omitted, does not use the value of DEFAULT_WIDTH, but instead uses the value zero.
- when WIDTH is zero (either explicitly or by default), reads until the syntax of an integer literal is not satisfied.

Implementation Guideline: This test should not raise DATA_ERROR.

Implementation Guideline: The test should check that reading occurs up to a period, up to the end of a line, and up to the closing # (or :) for a based literal.

- raises MODE_ERROR if the mode of the file is not IN_FILE.
- raises STATUS_ERROR if the file is not open (see T1).
- raises CONSTRAINT_ERROR if:
 - WIDTH is negative, or
 - WIDTH is greater than FIELD'LAST (only possible if FIELD'LAST < INTEGER'LAST), or
 - the value read is out of the range of the ITEM parameter, but within the range of the subtype used to instantiate INTEGER_IO.
- raises DATA_ERROR if:
 - the value read is not in the range of the subtype used to instantiate INTEGER_IO.
 - WIDTH is zero and:
 - the exponent is negative (reads to end of sequence of digits).
 - there is no closing # sign for a based literal (reads until syntax is violated).
 - for a based literal, the opening # is matched with a closing :, or vice versa (reading stops at : or #, respectively).
 - the letters in a based literal are out of the range of the base (reading stops at the end of the based literal).

- the base is not in the range 2 through 16 (reading stops at the end of the based literal).
- the base, the integer part, or the exponent part contains consecutive underscores (reading stops after the first underscore).
- there are consecutive underscores between digits of the base, the integer part, or the exponent (reading stops after the first underscore).
- there are leading or trailing underscores (reading stops before the leading underscore, and after the trailing underscore).
- the integer part of a based literal contains a character that is not a letter, digit, #, or : (reading stops before the offending character).
- a nondigit graphic character (other than a sign) is encountered before any digits have been read (reading stops before the nondigit character).

- WIDTH is nonzero and:

- all the cases for a zero WIDTH apply if WIDTH is large enough, except that reading stops after WIDTH characters have been read.

Implementation Guideline: Check that leading blanks on preceding lines are read only up to a line terminator if the number of blanks is less than WIDTH.

- reading stops before a complete literal has been read if WIDTH is small enough.
- the literal contains an embedded or trailing space or horizontal tabulation character (the tabulation character counts as a single character).
- the literal contains a decimal point.

- T5. Check that END_ERROR is raised (for GET from a file) if only the file terminator remains to be read (whether or not WIDTH is zero).

If WIDTH is zero, check that END_ERROR is raised if the only remaining characters in the file consist of line terminators, page terminators, spaces, and horizontal tabulation characters. After END_ERROR is raised, the file should be positioned before the file terminator.

Check that the last character in a file may be read without raising END_ERROR, and that after the last character of the file has been read, any attempt to read further characters will raise END_ERROR.

Check that END_ERROR is not raised when WIDTH > 0, fewer than WIDTH characters remain in the file, a based literal is being read, and the closing # (or :) has not yet been seen. (DATA_ERROR should be raised instead.)

Check that END_ERROR is not raised if fewer than WIDTH characters are remaining in the file and a real literal is being read.

- T6. Check that PUT to a file:

- correctly outputs the minimum and maximum integer values for an integer type as well as zero, for all bases 2 through 16.

- uses the default output file if no file is explicitly specified.
- uses the minimum field required if WIDTH is too small and the line length is sufficiently large.
- pads the output on the left with spaces if the value of WIDTH is greater than the minimum width required.
- has the effect of NEW_LINE as well as outputting the item when the number of characters to be output is less than the maximum line length, but when added to the current column number exceeds the maximum;
- uses the value of DEFAULT_WIDTH and DEFAULT_BASE when no explicit parameter is provided.
- raises MODE_ERROR if the mode of the file is not OUT_FILE.
- raises STATUS_ERROR if the file is not open (see T1).
- raises CONSTRAINT_ERROR if:
 - the specified value of BASE is not in the range 2 through 16.
 - the specified value of WIDTH is less than zero or greater than FIELD'LAST.
 - the value of ITEM is outside the range of the subtype used to instantiate INTEGER_IO.
- raises LAYOUT_ERROR when a nonzero line length, LL, is specified for the output file and:
 - WIDTH is greater than LL (since at least WIDTH characters must be output), or
 - the minimum width required for the output value is greater than LL.

Check that DEFAULT_BASE can only be assigned values in the range 2 through 16.

Check that DEFAULT_WIDTH can only be assigned non-negative values not exceeding FIELD'LAST.

T7. Check that GET from a string:

- correctly reads nonbased literals without an exponent when the value read is the minimum or maximum value for an Integer type.
Implementation Guideline: Use each of the predefined integer types.
- correctly reads a nonbased integer literal containing an exponent, whether or not the exponent is given in upper case, and whether or not a plus sign is explicitly provided in the exponent.
- correctly reads a based literal without an exponent, for bases 2 through 16, when the value read is the minimum or the maximum value for an integer type; check that any letters used may be in upper or lower case.
Implementation Guideline: Use each of the predefined integer types.
- correctly reads a based literal containing an exponent, for bases 2 through 16, independently of the case of the exponent or letters used in the literal, and whether or not a plus sign is explicitly provided in the exponent.

- raises **CONSTRAINT_ERROR** if the value read is out of the range of the **ITEM** parameter, but within the range of the subtype used to instantiate **INTEGER_IO**.
- raises **DATA_ERROR** if:
 - the value read is not in the range of the subtype used to instantiate **INTEGER_IO**.
 - the exponent is negative.
 - there is no closing # sign for a based literal.
 - for a based literal, the opening # is matched with a closing :, or vice versa.
 - the letters in a based literal are out of the range of the base.
 - the base is not in the range 2 through 16.
 - the base, integer part, or exponent part contains consecutive underscores.
 - there are leading or trailing underscores.
 - the integer part of a based literal contains a character that is not a letter, digit, #, or :.
 - a nondigit graphic character (other than a sign) is encountered before any digits.
 - the literal contains an embedded or trailing space or a horizontal tabulation character.
- raises **END_ERROR** if an attempt is made to skip the end of the string, i.e., if the string is null or only contains spaces and/or horizontal tabulation characters.

Check that **LAST** contains the index value of the last character read from the string if no exception is raised.

T8. Check that **PUT** to a string:

- correctly outputs the minimum and maximum integer values for an integer type as well as zero, for all bases 2 through 16.
- pads the output on the left with spaces if the length of the string is greater than the minimum width required.
- uses the value of **DEFAULT_BASE** when no explicit parameter is provided.
- raises **CONSTRAINT_ERROR** if:
 - the specified value of **BASE** is not in the range 2 through 16.
 - the value of **ITEM** is outside the range of the subtype used to instantiate **INTEGER_IO**.
- raises **LAYOUT_ERROR** when the minimum width required for the output value is greater than the length of the string.

T9. Check the names of the formal parameters.

14.3.8 Input-Output for Real Types

Semantic Ramifications

S1. Note that a *blank* is defined as a space or a horizontal tabulation character (RM 14.3.5/5). Hence, leading spaces or tabs are skipped when reading a real literal.

S2. When reading with a WIDTH of zero, reading stops when the initial sequence of the syntax of a real literal is no longer satisfied (RM 14.3.5/5, /6). RM 2.4/1 defines a real literal to be a numeric literal with a decimal point. Hence, when reading the following input:

```
14.3 X      -- next character read is X
14.3EX      -- next character read is X
14E-3X      -- next character read is E
```

DATA_ERROR is raised by GET and the next character to be read is indicated by the comments. Since lexical elements cannot extend across line boundaries (RM 2.2/1, /2), and since GET analyzes input sequences of characters as lexical elements (RM 14.3/2), reading of a numeric literal stops when a line terminator is encountered.

S3. When WIDTH is nonzero, exactly WIDTH characters are read (unless a line terminator is seen first; then only characters preceding the line terminator are read):

```
14.3e3XYZ
```

Calling GET with a WIDTH of 7 causes 7 characters to be read. DATA_ERROR is raised since "14.3e3X" does not satisfy the syntax of a numeric literal. Note that DATA_ERROR would also be raised if X were a space character. DATA_ERROR will not be raised if X is replaced with a line terminator, in which case, only 6 characters will be read.

S4. A real literal must have an embedded decimal point (RM 2.4) and so is at least three characters long. Hence if WIDTH is 1 or 2, GET will raise DATA_ERROR.

S5. The syntax for a based real literal (RM 2.4.2/2) permits any letter to be used in the part enclosed in # characters. Therefore, if the input is:

```
8#16.8ZF#B      -- input data to be read
```

all characters up to the second # will be read before DATA_ERROR is raised. (DATA_ERROR will be raised since 8, Z, and F are all incompatible with the base.)

S6. The letter E in the exponent part and the letters used as extended digits in based notation may appear in upper or lower case on input (RM 2.4.1/3); RM 14.3.8/2 specifies that a capital "E" is to be used on output.

S7. Numeric literals input in based notation may have the # character replaced uniformly (within a literal) by : characters (RM 2.10); the implementation has the same freedom for output. In particular, if the compiler runs in an environment supporting the # character, but the object code runs in an environment supporting only the colon, then different forms of literals will be used for input-output vs. source code.

S8. Note that an attempt to read too large a value raises DATA_ERROR or CONSTRAINT_ERROR, but not NUMERIC_ERROR:

```
subtype ONE is FLOAT range 0.0 .. 1.0;
package FLT_IO is new FLOAT_IO(ONE);
ITEM : ONE range 0.0 .. 0.5;
BIG  : FLOAT;
LAST : POSITIVE;
```

```

...
FLT_IO.GET ("2.0", ITEM, LAST);           -- DATA_ERROR
FLT_IO.GET ("2.0", BIG, LAST);           -- DATA_ERROR
FLT_IO.GET ("1.0E10 000 000 000 000", ITEM, LAST); -- DATA_ERROR
FLT_IO.GET ("0.75", ITEM, LAST);         -- CONSTRAINT_ERROR

```

The first three calls raise `DATA_ERROR` because the values being input exceed the range of the subtype used to instantiate `FLT_IO`. Note that in the second case, `DATA_ERROR` is raised even though 2.0 belongs to `BIG`'s subtype. The last call raises `CONSTRAINT_ERROR` because 0.75 in `ONE` is `TRUE`, i.e., 0.75 belongs to the subtype used to instantiate `FLOAT_IO`, but 0.75 exceeds 0.5, the upper bound for `ITEM`, which is the actual parameter used in the `GET` call. In short, `DATA_ERROR` is raised if the input value lies outside the range of the subtype used in an instantiation. `CONSTRAINT_ERROR` is raised if the input value lies inside the range of the subtype used in the instantiation, but outside the range of the actual parameter's subtype.

S9. The RM says `GET` returns the value that corresponds to the literal that is read (RM 14.3.8/9). This means converting the mathematical (i.e., *universal_real*) value of the literal to `ITEM`'s type. Such conversions are governed by the rules of RM 4.5.7, since conversion is a basic operation. Hence, if a literal's value is in the range of safe numbers for `ITEM`'s base type, and the literal's value is also the value of a safe number for `ITEM`'s type, then the literal must be converted to the corresponding safe number, i.e., exact conversion is required. Literals whose value lies between safe numbers must be converted to a value bounded by consecutive safe numbers.

S10. The RM places no restrictions on the accuracy of output conversion. Our tests will check to see if an implementation satisfies the following rules:

- for floating point types and certain fixed point types (e.g., types whose `SMALL` is a power of two), `PUT` outputs `ITEM`'s value exactly if `AFT` is sufficiently long. (Note that every safe number of these types can be represented exactly as a decimal number, since safe numbers are represented in base two.)
- if `AFT` is not sufficiently long to permit `ITEM`'s value to be represented exactly, `PUT` outputs the decimal value closest to `ITEM`'s actual value. When two decimal values are equally close to `ITEM`'s actual value, either one may be output. However, programmers will be less surprised if the value furthest from zero is chosen in this case (e.g., when `AFT` = 1, -1.25 should be output as -1.3, and 1.25 should be output as 1.3; note that the RM, however, explicitly allows -1.2 and 1.2 to be output in these cases).

S11. The minimum length needed to output a real number to a file depends on the value being output and the values of the `FORE`, `AFT`, and `EXP` parameters. Note that a sign position is needed only for negative numbers. The number of digits needed before the decimal point depends on the value of the `EXP` parameter, namely, if `EXP` is greater than zero, then the number needed is 1 (exclusive of sign). If `EXP` is zero, then the number of digits depends on the absolute magnitude of `ITEM`'s value. The minimum number of digits is 1 if $\text{abs}(\text{ITEM}) < 1.0$. Otherwise, the minimum number of digits preceding the decimal point is:

$$\text{floor}(\log_{10}(\text{abs}(\text{ITEM}))) + 1,$$

where the `floor` function yields the largest integer value less than or equal to its argument. Hence, $\text{floor}(1.9) = 1$, and $\text{floor}(2.0) = 2$. Some values yielded by this expression are:

ITEM	digits
10	2
9	1
0.1	0
0.5	0

The expression `BOOLEAN'POS(ITEM < 0.0)` yields 1 if ITEM is negative; otherwise zero. Hence, the minimum number of nonblank characters preceding the decimal point when `EXP = 0` and `ITEM /= 0.0` is given by:

$$\max(1, \text{floor}(\log_{10}(\text{abs}(\text{ITEM}))) + 1) +$$

$$\text{BOOLEAN'POS}(\text{ITEM} < 0.0) \text{ when } \text{abs}(\text{ITEM}) \geq 1.0$$

S12. When `EXP` is nonzero, then the minimum number of characters preceding the decimal point is one plus a position for a minus sign if the number is negative:

$$1 + \text{BOOLEAN'POS}(\text{ITEM} < 0.0)$$

The number of digits in the fraction part is:

$$\max(1, \text{AFT})$$

The number of digits in the exponent value is given by:

$$\text{exp_digits} = \text{floor}(\log_{10}(\text{abs}(\text{exponent_value}))) + 1$$

where `exponent_value =`

$$1 \text{ if } \text{ITEM} = 0.0 \text{ and otherwise equals}$$

$$\text{floor}(\log_{10}(\text{abs}(\text{ITEM})))$$

When `EXP` is nonzero, its value includes a position for the exponent sign. Consequently, the number of digits in an exponent part is:

$$\max(1, \text{EXP}-1, \text{exp_digits})$$

If `EXP-1` is greater than `exp_digits`, then leading zeroes are used.

S13. The value of `FORE` determines a minimum number of characters that precede the decimal point. Therefore, the final formulas for the minimum field size needed to represent a real value are as follows (where `sign = BOOLEAN'POS(ITEM < 0.0)` and `ITEM /= 0.0`):

when `EXP = 0`:

```

sign                -- minus sign if necessary
+ max(1, FORE-sign, floor(log10(abs(ITEM))) + 1)
                    -- characters in integer part
+ 1                  -- decimal point
+ max(1, AFT)        -- number of digits in fractional part

```

when `EXP > 0`:

```

sign                -- minus sign if necessary
+ max(1, FORE-sign) -- integer part
+ 1                  -- decimal point
+ max(1, AFT)        -- number of digits in fractional part
+ 2                  -- the "E" and the exponent's sign

```

+ $\max(1, \text{EXP}-1, \text{exp_digits})$ -- number of exponent digits

Note that at least one digit is always output before the decimal point. Hence, although a value of zero is allowed for FORE, it is always ignored. Since `exp_digits` has a minimum value of one when `EXP > 0`, and at least one fractional digit is required, the minimum number of characters needed to output a real number is 3 when `EXP` equals zero, and 6 when `EXP` is greater than zero. These minimums are increased by 1 when the value being output is negative.

Changes from July 1982

S14. When width is nonzero, leading line terminators are not skipped. Instead, reading stops when a line terminator is seen or when `WIDTH` characters have been read, whichever comes first. This means that reading does not necessarily stop when the syntax of a numeric literal is violated.

S15. One fractional digit is output when `AFT` is zero.

S16. When outputting 0.0 with an exponent, the exponent part is specified to equal zero. (Previously the exponent value was implementation defined in this case.)

S17. GET from a STRING no longer replaces the value of `LAST` with an "undefined" value when an exception is raised. Instead, the normal semantics of out parameters defines the effect on `LAST`, i.e., the value of `LAST` is unchanged (since `LAST` is a scalar parameter). The July 1982 wording implied that subsequent accesses to `LAST` could be erroneous (see RM 3.2.1/18).

Changes from July 1980

S18. PUT has the same form for both fixed and floating point types, and the name and meaning of some formal parameters is changed.

S19. GET has an optional `WIDTH` parameter.

S20. Default parameter values are defined by variables declared in the generic package; hence, these defaults can be changed dynamically during program execution.

S21. GET and PUT are defined for STRINGs as well as files.

S22. DATA_ERROR is raised by GET (instead of CONSTRAINT_ERROR) if the value read is not in the range of the subtype used to instantiate the package.

Exception Conditions

E1 Exceptions (file input)

E1.1 STATUS_ERROR is raised if the file is not open.

E1.2 MODE_ERROR is raised if the file's mode is not `IN_FILE`.

E1.3 DATA_ERROR is raised if:

- the input does not have the syntax of a numeric literal with a decimal point.
- the number being input lies outside the range specified for the subtype used in the instantiation of `FLOAT_IO` or `FIXED_IO`.

E4 CONSTRAINT_ERROR is raised if:

- the `WIDTH` parameter has a negative value or exceeds `FIELD'LAST` (and `FIELD'LAST < INTEGER'LAST`).
- the value read does not satisfy the actual parameter's range, and the actual parameter's range is a subset of the range of the subtype used to instantiate `FLOAT_IO` or `FIXED_IO`.

E5. **END_ERROR** is raised if:

- **WIDTH** is zero and the only characters read are spaces, horizontal tabulation characters, line terminators, and page terminators or,
- **WIDTH** is nonzero, and the file is positioned just before the file terminator.

GET Exceptions (string Input)

E6. **DATA_ERROR** is raised if:

- the input does not have the syntax of a numeric literal with a decimal point.
- the number being input lies outside the range specified for the subtype used in the instantiation of **FLOAT_IO** or **FIXED_IO**.

E7. **CONSTRAINT_ERROR** is raised if the value read does not satisfy the actual parameter's range, and the actual parameter's range is a subset of the range of the subtype used to instantiate **FLOAT_IO** or **FIXED_IO**.

E8. **END_ERROR** is raised if the only characters read are spaces, horizontal tabulation characters, line terminators, and page terminators, or if the input string is null.

PUT Exceptions (file output)

E9. **STATUS_ERROR** is raised if the file is not open.

E10. **MODE_ERROR** is raised if the file's mode is not **IN_FILE**.

E11. **LAYOUT_ERROR** is raised (RM 14.3.5/10) if the line length (**LL**) for the file is fixed (i.e., not zero), and:

- $LL < \max(\text{FORE}, 1) + 1 + \max(\text{AFT}, 1) + \text{EXP}$
- when $\text{EXP} = 0$ and $\text{abs}(\text{ITEM}) \neq 0.0$:

$$LL < \text{sign} + \max(1, \text{FORE} - \text{sign}, \text{floor}(\log_{10}(\text{abs}(\text{ITEM}))) + 1) + 1 + \max(1, \text{AFT})$$
- when $\text{EXP} > 0$ and $\text{ITEM} \neq 0.0$:

$$LL < \text{sign} + \max(1, \text{FORE} - \text{sign}) + 1 + \max(1, \text{AFT}) + 2 + \max(1, \text{EXP} - 1, \text{exp_digits})$$
- when $\text{EXP} = 0$ and $\text{ITEM} = 0.0$:

$$LL < 3$$
- when $\text{EXP} > 0$ and $\text{ITEM} = 0.0$:

$$LL < 4 + \max(2, \text{EXP})$$

E12. **CONSTRAINT_ERROR** is raised:

- if the value of **ITEM** lies outside the range of the subtype used to instantiate **FLOAT_IO** or **FIXED_IO**.
- when the value specified for **FORE**, **AFT**, or **EXP** is negative or exceeds **FIELD'LAST**, and **FIELD'LAST** < **INTEGER'LAST**.

PUT Exceptions (string output)

E13. **LAYOUT_ERROR** is raised if the length of the string variable is less than the minimum required length for the value being output (RM 14.3.5/10).

E14. **CONSTRAINT_ERROR** is raised if the value of **ITEM** lies outside the range of the subtype used to instantiate **FLOAT_IO** or **FIXED_IO**.

Test Objectives and Design Guidelines

- T1. Check that each operation raises **STATUS_ERROR** if the file is not open.
- T2. Check that **FLOAT_IO** and **FIXED_IO** can be instantiated for predefined types and for user-defined real types.

Check that **FLOAT_IO** cannot be instantiated with a nonfloat type.

Check that **FIXED_IO** cannot be instantiated with a nonfixed type.

- T3. Check that **PUT** and **GET** for fixed and float types are not available without instantiating **FIXED_IO** and **FLOAT_IO**, respectively.

- T4. Check that **GET** from a file, for both fixed and float types:

- correctly reads nonbased literals without an exponent when the value read is a positive, negative, and zero real literal.
- correctly reads a nonbased literal containing an exponent, whether or not the exponent is given in upper case, and whether or not a sign (plus or minus) is explicitly provided in the exponent.
- correctly reads a based literal without an exponent, for bases 2 through 16, when the value read is a positive or a negative value; check that any letters used may be in upper or lower case.
- correctly reads a based literal containing an exponent, for bases 2 through 16, independently of the case of the exponent or letters used in the literal, and whether or not a sign (plus or minus) is explicitly provided in the exponent.
- allows **:** instead of **#** in based literals.
- uses the default input file if no file is explicitly specified.
- skips leading spaces, horizontal tabulation characters, line terminators, and page terminators if **WIDTH** is zero.
- when **WIDTH** is nonzero, reads at most **WIDTH** characters or up to the next line terminator; counts leading spaces and horizontal tabulation characters as part of the **WIDTH**.
- when **WIDTH** is omitted, uses the value zero.
- when **WIDTH** is zero (either explicitly or by default), reads until the syntax of a real literal is not satisfied.

Implementation Guideline: This test should not raise **DATA_ERROR**.

Implementation Guideline: The test should check that reading occurs up to the end of a sequence of digits following a decimal point, or up to the end of a line, and up to the closing **#** (or **:**) for a based literal.

- raises **MODE_ERROR** if the mode of the file is not **IN_FILE**.
- raises **STATUS_ERROR** if the file is not open (see T1).

- raises `CONSTRAINT_ERROR` if:
 - `WIDTH` is negative, or
 - `WIDTH` is greater than `FIELD'LAST` (only possible if `FIELD'LAST` < `INTEGER'LAST`), or
 - the value read is out of the range of the `ITEM` parameter, but within the range of the subtype used to instantiate `FLOAT_IO` or `FIXED_IO`.
- raises `DATA_ERROR` if:
 - the value read is not in the range of the subtype used to instantiate `FIXED_IO` or `FLOAT_IO`.
 - `WIDTH` is zero and:
 - no decimal point is present (reading stops after a sequence of digits; for based literals, reading stops after a sequence of digits and letters).
 - the decimal point is not followed by a digit or, for based literals, a letter (reading stops after the decimal point).
 - the decimal point is not preceded by a digit (or a letter in a based literal) (reading stops before the decimal point).
 - there is no closing `#` sign for a based literal (reads until syntax is violated).
 - for a based literal, the opening `#` is matched with a closing `:`, or vice versa (reading stops at `:` or `#`, respectively).
 - the letters in a based literal are out of the range of the base (reading stops at the end of the based literal).
 - the base is not in the range 2 through 16 (reading stops at the end of the based literal).
 - the base, integer part, or exponent part contain consecutive underscores (reading stops after the first underscore).
 - there are leading or trailing underscores (reading stops before the leading underscore, and after the trailing underscore).
 - the integer or fraction part of a based literal contains a character that is not a letter, digit, `#`, or `:` (reading stops before the offending character).
 - a nondigit graphic character (other than a sign) is encountered before any digits have been read (reading stops before the nondigit character).
 - `WIDTH` is nonzero and:
 - all the cases for a zero `WIDTH` apply if `WIDTH` is large enough, except that reading stops after `WIDTH` characters have been read.
Implementation Guideline: Check that leading blanks on preceding lines are read only up to a line terminator if the number of blanks is less than `WIDTH`.
Implementation Guideline: If no line terminator is encountered, check that

WIDTH characters are read, even if the syntax for a real literal is not satisfied by the sequence of characters.

- reading stops before a complete literal has been read if WIDTH is small enough.
- the literal contains an embedded or trailing space or horizontal tabulation character.

T5. Check that END_ERROR is raised (for GET from a file) if only the file terminator remains to be read (whether or not WIDTH is zero).

If WIDTH is zero, check that END_ERROR is raised if the only remaining characters in the file consist of line terminators, page terminators, spaces, and horizontal tabulation characters. After END_ERROR is raised, the file should be positioned before the file terminator.

Check that the last character in a file may be read without raising END_ERROR, and that after the last character of the file has been read, any attempt to read further characters will raise END_ERROR.

Check that END_ERROR is not raised if fewer than WIDTH characters are remaining in the file and a real literal is being read. (DATA_ERROR should be raised instead, if any exception at all is raised.)

T6. Check that PUT to a file (for fixed or float values):

- may be called with or without a FORE, EXP, and AFT parameter, the correct default values are used, and the correct formatted output is obtained. In particular, check that PUT:
 - will output the whole converted string if the integer part of the converted number exceeds the value of FORE, or the required exponent length exceeds EXP (when EXP > 0).
Implementation Guideline: Include a value of zero for FORE.
 - will insert a sufficient number of leading spaces if a FORE parameter was given and is greater than the length of the converted number's required integer part.
 - will insert a sufficient number of leading zeroes if an EXP parameter was given and is greater than the length of the converted number's required exponent part.
Implementation Guideline: Check that the number of spaces output when FORE is too big is not affected when EXP is too small.
 - outputs one fractional digit even when AFT is zero.
 - outputs at least one digit in the exponent if EXP is one.
 - generates output conforming to the proper syntax, including a minus sign if the value is negative.
 - uses appropriate values of DEFAULT_FORE, DEFAULT_AFT, or DEFAULT_EXP if these parameters are omitted.
- uses the default output file if no file is explicitly specified.
- has the effect of NEW_LINE as well as outputting the item when the number of characters to be output is less than the maximum line length, but when added to the current column number exceeds the maximum.

- raises `MODE_ERROR` if the mode of the file is not `OUT_FILE`.
- raises `STATUS_ERROR` if the file is not open (see T1).
- raises `CONSTRAINT_ERROR` if:
 - the specified value of `FORE`, `AFT`, or `EXP` is less than zero or greater than `FIELD'LAST`.
 - the value of `ITEM` is outside the range of the type used to instantiate `FIXED_IO` or `FLOAT_IO`.
- raises `LAYOUT_ERROR` when a nonzero line length, `LL`, is specified for the output file and:
 - $\text{FORE} + \text{AFT} + 1 > \text{LL}$ when `FORE` and `AFT` are both nonzero and `EXP` = 0.
 - $\text{FORE} + \text{AFT} + 2 > \text{LL}$ when either `FORE` or `AFT` are zero (but not both) and `EXP` = 0.
 - $3 > \text{LL}$ when `FORE`, `AFT`, and `EXP` are all zero.
 - $\text{FORE} + \text{AFT} + 2 + \max(2, \text{EXP}) > \text{LL}$ when `FORE`, `AFT`, and `EXP` are all nonzero.
 - $\text{FORE} + \text{AFT} + 3 + \max(2, \text{EXP}) > \text{LL}$ when either `FORE` or `AFT` are zero (but not both) and `EXP` is nonzero.
 - $\max(2, \text{EXP}) + 4 > \text{LL}$ when `FORE` and `AFT` are both zero and `EXP` is nonzero.
 - the minimum width required for the output value is greater than `LL` and the values for `FORE`, `AFT`, and `EXP` do not satisfy one of the relationships given above.

Implementation Guideline: If `LAYOUT_ERROR` is raised, no characters should be output.

- will output zero as 0.0 (when `EXP` = 0) and 0.0E+0 (when `EXP` = 1 or 2).

Implementation Guideline: Also check for `EXP` > 2.

Check that `DEFAULT_FORE`, `DEFAULT_AFT`, and `DEFAULT_EXP` can only be assigned non-negative values not exceeding `FIELD'LAST`.

T9. Check that GET from a string, for both fixed and float types:

- correctly reads nonbased literals without an exponent when the value read is a positive, negative, and zero real literal.
- correctly reads a nonbased literal containing an exponent, whether or not the exponent is given in upper case, and whether or not a sign (plus or minus) is explicitly provided in the exponent.
- correctly reads a based literal without an exponent, for bases 2 through 16, when the value read is a positive or negative value; check that any letters used may be in upper or lower case.
- correctly reads a based literal containing an exponent, for bases 2 through 16, independently of the case of the exponent or letters used in the literal, and whether or not a sign (plus or minus) is explicitly provided in the exponent.
- allows : instead of # in based literals.

- ignores leading space and horizontal tabulation characters.
- raises `CONSTRAINT_ERROR` if the value read is out of the range of the `ITEM` parameter, but within the range of the subtype used to instantiate `FLOAT_IO` or `FIXED_IO`.
- raises `DATA_ERROR` if:
 - the value read is not in the range of the subtype used to instantiate `FIXED_IO` or `FLOAT_IO`.
 - no decimal point is present.
 - the decimal point is not followed by a digit or, for based literals, a letter.
 - the decimal point is not preceded by a digit (or letter, in a based literal).
 - there is no closing `#` sign for a based literal.
 - for a based literal, the opening `#` is matched with a closing `:`, or vice versa.
 - the letters in a based literal are out of the range of the base.
 - the base is not in the range 2 through 16.
 - the base, integer part, or exponent part contain consecutive underscores.
 - there are leading or trailing underscores.
 - the integer or fraction part of a based literal contain a character that is not a letter, digit, `#`, or `:`.
 - a nondigit graphic character (other than a sign) is encountered before any digits have been read.
 - the literal contains an embedded or trailing space or horizontal tabulation character.
- raises `END_ERROR` if the string is null or contains only spaces and/or horizontal tabulation characters.

Check that `LAST` contains the index value of the last character read from the string.

T10. Check that `PUT` to a string (for fixed or float values):

- may be called with or without an `AFT` and `EXP` parameter, the correct default values are used, and the correct formatted output is obtained. In particular, check that `PUT`:
 - will output the whole converted string if the required exponent length exceeds `EXP` (when `EXP > 0`).
 - will insert a sufficient number of leading spaces if the minimum required length is less than the length of the string.
 - outputs one fractional digit even when `AFT` is zero.
 - outputs at least one digit in the exponent if `EXP` is one.
 - generates output conforming to the proper syntax, including a minus sign if the value is negative.

- uses appropriate values of `DEFAULT_AFT` or `DEFAULT_EXP` if the `AFT` or `EXP` parameters are omitted.
 - raises `CONSTRAINT_ERROR` if:
 - the specified value of `AFT` or `EXP` is less than zero or greater than `FIELD'LAST`.
 - the value of `ITEM` is outside the range of the subtype used to instantiate `FIXED_IO` or `FLOAT_IO`.
 - raises `LAYOUT_ERROR` when the length of the string is `LL` and:
 - $AFT + 2 > LL$ when `AFT` is nonzero and `EXP = 0`.
 - $3 > LL$ when `AFT` and `EXP` are both zero.
 - $AFT + 3 + \max(2, EXP) > LL$ when `AFT` and `EXP` are both nonzero.
 - $4 + \max(2, EXP) > LL$ when `AFT` is zero and `EXP` is nonzero.
 - the minimum width required for the output value is greater than `LL` and the values for `AFT` and `EXP` do not satisfy one of the relationships given above.
- Implementation Guideline:* If `LAYOUT_ERROR` is raised, no characters should be output.
- will output zero as `0.0` (when `EXP = 0`) and `0.0E+0` (when `EXP = 1` or `2`).

T11. Check that floating point safe numbers are input exactly.

Implementation Guideline: Use values such as `'SMALL` and `'LARGE`. Check for a variety of floating point types, including the most precise type.

Implementation Guideline: Use string input as well as file input.

Check that numbers lying between model numbers are input with the same accuracy as the corresponding literals.

T12. Check that safe floating point numbers are output exactly if `AFT` is sufficiently large.

Implementation Guideline: Note: `AFT` cannot be greater than `FIELD'LAST`.

Check that numbers are rounded correctly on output.

Implementation Guideline: Include positive and negative values, and some values whose rounding error is slightly above, slightly below, and exactly one-half the value of the last output position.

Implementation Guideline: Use both string and file output.

T13. Check that fixed point model numbers are input exactly for both string and file input.

Implementation Guideline: Use values such as `'SMALL` and `'LARGE`. Check for a variety of fixed point types, including some types for which `'SMALL` has been specified as other than a power of two. (Note: it is not required that all implementations support an explicit specification of `'SMALL` via a length clause; see IG 13.2/S).

Check that numbers lying between two fixed point model numbers are input with the same accuracy as the corresponding literals.

T14. Check that safe fixed point numbers are output exactly if `AFT` is sufficiently large and the value is exactly representable as a decimal literal.

Check that fixed point values are rounded correctly on output.

Implementation Guideline: See Guidelines for T12.

T15. Check the names of the formal parameters.

14.3.9 Input-Output for Enumeration Types

Semantic Ramifications

S1. Note that PUT and GET exist for the CHARACTER data type without instantiating ENUMERATION_IO, but the semantic effect of TEXT_IO's PUT and GET for CHARACTERs is different from the effect when ENUMERATION_IO is instantiated with the CHARACTER type (see IG 14.3.6/S).

S2. Note that for an enumeration literal that is an identifier, the output case is independent of the way the identifier appeared in the type declaration. In particular, even if the declaration used mixed case, the output will be all lower or all upper case.

S3. Note that DATA_ERROR is raised both for literals that do not belong to the base type and for out of range values. For example:

```

subtype HEX      is CHARACTER range 'A'..'F';
type    HEX2     is ('A', 'B', 'C', 'D', 'E', 'F');
package HEX_IO   is new ENUMERATION_IO(HEX);
package HEX2_IO  is new ENUMERATION_IO(HEX2);

subtype HEX2S    is HEX2 range 'A'..'C';
package HEX2S_IO is new ENUMERATION_IO(HEX2S);

X      : HEX;
X2S    : HEX2S;
```

If the input contains 'Z', HEX_IO.GET(X) raises DATA_ERROR (as would HEX2_IO.GET(X2S)), not CONSTRAINT_ERROR (as one might expect). If the input contains 'F', however, HEX2_IO.GET(X2S) will raise CONSTRAINT_ERROR, and HEX2S_IO.GET(X2S) will raise DATA_ERROR.

S4. The RM says that the effect of instantiating ENUMERATION_IO with an integer type is undefined. An implementation can check to see if ENUMERATION_IO has been instantiated with an integer type by using the following code:

```

declare
  VAL : constant STRING := ENUM' IMAGE (ENUM' VAL(0));
begin
  if VAL(1) = ' ' or VAL(1) = '-' then
    raise USE_ERROR;    -- integer type used
  end if;
end;
```

Note that no enumeration literal can begin with either a space or a minus sign. Moreover, zero is a valid position number for every enumeration and integer type.

Changes from July 1982

S5. The case of the output is specified with the enumeration type TYPE_SET rather than with a BOOLEAN value.

S6. The variable controlling the default output case is named DEFAULT_SETTING instead of DEFAULT_IS_LC.

S7. PUT's formal parameter controlling the case of the output is called SET instead of LC.

Changes from July 1980

- S8. Variables controlling the default values for width and case are provided.
- S9. GET skips leading blanks, line terminators, and page terminators.
- S10. Additional GET and PUT procedures are specified that operate on STRINGS rather than files.

Exception Conditions**GET Exceptions (file input)**

- E1. STATUS_ERROR is raised if the file is not open.
- E2. MODE_ERROR is raised if the file's mode is not IN_FILE.
- E3. DATA_ERROR is raised if:
 - the input does not have the syntax of a character literal or identifier.
 - the literal or identifier is not a value of the base type used to instantiate ENUMERATION_IO.
 - the literal being input lies outside the range specified for the subtype used in the instantiation of ENUMERATION_IO.
- E4. CONSTRAINT_ERROR is raised if the value read does not satisfy the actual parameter's range and the actual parameter's range is a subset of the range of the subtype used to instantiate ENUMERATION_IO.
- E5. END_ERROR is raised if the only characters read are spaces, horizontal tabulation characters, line terminators, and page terminators.

GET Exceptions (string input)

- E6. DATA_ERROR is raised if:
 - the input does not have the syntax of a character literal or identifier
 - the literal or identifier is not a value of the base type used to instantiate ENUMERATION_IO.
 - the value being input lies outside the range specified for the subtype used in the instantiation of ENUMERATION_IO.
- E7. CONSTRAINT_ERROR is raised if the value read does not satisfy the actual parameter's range and the actual parameter's range is a subset of the range of the subtype used to instantiate ENUMERATION_IO.
- E8. END_ERROR is raised if the only characters read are spaces and horizontal tabulation characters, or if the input string is null.

PUT Exceptions (file output)

- E9. STATUS_ERROR is raised if the file is not open.
- E10. MODE_ERROR is raised if the file's mode is not IN_FILE.
- E11. LAYOUT_ERROR is raised (RM 14.3.5/10) if the line length (LL) for the file is fixed (i.e., not zero), and:

- $LL < \text{the length of the identifier or character literal to be output (whether or not WIDTH is zero), or}$
- $WIDTH > LL$.

E12. **CONSTRAINT_ERROR** is raised:

- if the value of **ITEM** lies outside the range of the subtype used to instantiate **ENUMERATION_IO**.
- when the value specified for **WIDTH** is negative or exceeds **FIELD'LAST**, and **FIELD'LAST** < **INTEGER'LAST**.

PUT Exceptions (string output)

- E13. **LAYOUT_ERROR** is raised if the length of the string variable is less than the minimum required length for the value being output (RM 14.3.5/10).
- E14. **CONSTRAINT_ERROR** is raised if the value of **ITEM** lies outside the range of the subtype used to instantiate **ENUMERATION_IO**.

Test Objectives and Design Guidelines

- T1. Check that **GET** and **PUT** raise **STATUS_ERROR** if the file is not open.
- T2. Check that **ENUMERATION_IO** can be instantiated for predefined and user-defined enumeration types.
- Implementation Guideline:* Include an instantiation for **BOOLEAN**.
- T3. Check that **GET** and **PUT** for enumeration types (other than character; see RM 14.3.6) are not available without instantiating **ENUMERATION_IO**.
- T4. For **GET** from a file, check that the last nonblank character in a file may be read without raising **END_ERROR**, and that after the last character of the file has been read, any attempt to read further characters will raise **END_ERROR**.

Check that **END_ERROR** is raised by **GET** when the only remaining characters in a file are spaces, tabulation characters, line terminators, and page terminators.

T5. Check that **GET** from a file:

- correctly reads identifiers and character literals, i.e., distinguishes the case in character literals but not in identifiers.

Implementation Guideline: Use the predefined **BOOLEAN** and **CHARACTER** types as well as user-defined types.

- reads until the syntax for an identifier or character literal is not satisfied.

Implementation Guideline: This check should not raise **DATA_ERROR**.

Implementation Guideline: A character literal should be followed by an apostrophe; an identifier should be terminated by a line terminator and by a character that is not a letter, digit, or underscore.

- uses the default input file if no file is explicitly specified.
- skips leading spaces, horizontal tabulation characters, line terminators, and page terminators.
- raises **MODE_ERROR** if the mode of the file is not **IN_FILE**.
- raises **STATUS_ERROR** if the file is not open (see T1).

- raises `CONSTRAINT_ERROR` if the value read is out of the range of the `ITEM` parameter, but within the range of the subtype used to instantiate `ENUMERATION_IO`.
- raises `DATA_ERROR` if:
 - the lexical element retrieved is not a value of the enumeration type.
 - the lexical element is not in the range of the subtype used to instantiate `ENUMERATION_IO`.
 - the lexical element is an identifier terminated with two underscores (reading stops after the first underscore).
 - the input is a character literal, but with the terminating apostrophe omitted.

T6. Check that PUT to a file:

- outputs identifiers in lower case if `LOWER_CASE` is specified explicitly or as the value of `DEFAULT_SETTING`, but performs no conversion on character literals.
- outputs identifiers in upper case if `UPPER_CASE` is specified explicitly or as the value of `DEFAULT_SETTING`, but performs no conversion on character literals.
- outputs character literals between single quote marks.
- uses the minimum field required if `WIDTH` is too small or zero, and the line length is sufficiently large.
- pads the output on the right with spaces if the value of `WIDTH` is greater than the minimum width required.
- has the effect of `NEW_LINE` as well as the effect of outputting the item when the number of characters to be output is less than the maximum line length, but when added to the current column number exceeds the maximum.
- uses the value of `DEFAULT_WIDTH` and `DEFAULT_SETTING` when no explicit parameter is provided.
- uses the default output file if no file is explicitly specified.
- raises `MODE_ERROR` if the mode of the file is not `OUT_FILE`.
- raises `STATUS_ERROR` if the file is not open (see T1).
- raises `CONSTRAINT_ERROR` if:
 - the specified value of `WIDTH` is less than zero or greater than `FIELD'LAST`.
 - the value of `ITEM` is outside the range of the subtype used to instantiate `ENUMERATION_IO`.
- raises `LAYOUT_ERROR` when a nonzero line length, `LL`, is specified for the output file and:
 - `WIDTH` is greater than `LL` (since at least `WIDTH` characters must be output), or

- the minimum width required for the output value is greater than LL.

Check that `DEFAULT_WIDTH` can only be assigned non-negative values not exceeding `FIELD_LAST`.

T7. Check that PUT to a string:

- outputs identifiers in lower case if `LOWER_CASE` is specified explicitly or as the value of `DEFAULT_SETTING`, but performs no conversion on character literals.
- outputs identifiers in upper case if `UPPER_CASE` is specified explicitly or as the value of `DEFAULT_SETTING`, but performs no conversion on character literals.
- outputs character literals between single quote marks.
- pads the output on the right with spaces if the length of the string is greater than the minimum width required.
- uses the value of `DEFAULT_SETTING` when no explicit parameter is provided.
- raises `CONSTRAINT_ERROR` if the value of `ITEM` is outside the range of the subtype used to instantiate `ENUMERATION_IO`.
- raises `LAYOUT_ERROR` when the minimum width required for the output value is greater than the length of the string.

T8. Check that GET from a string:

- correctly reads identifiers and character literals, i.e., distinguishes the case in character literals, but not in identifiers.
Implementation Guideline: Use the predefined `BOOLEAN` and `CHARACTER` types as well as user-defined types.
- skips leading spaces and horizontal tabulation characters.
- reads until the syntax of an identifier or a character literal is no longer satisfied.
- sets `LAST` to the correct value.
- raises `CONSTRAINT_ERROR` if the value read is out of the range of the `ITEM` parameter, but within the range of the subtype used to instantiate `ENUMERATION_IO`.
- raises `DATA_ERROR` if:
 - the lexical element retrieved is not a value of the enumeration type.
 - the lexical element is not in the range of the subtype used to instantiate `ENUMERATION_IO`.
 - the lexical element is an identifier terminated with two underscores (reading stops after the first underscore).
 - the input is a character literal, but with the terminating apostrophe omitted.
- raises `END_ERROR` if the input string is null or contains only spaces and horizontal tabulation characters.

T9. Check the names of the formal parameters.

14.3.10 Specification of the Package Text_IO

The implications of the TEXT_IO package have been discussed in IG 14.3 and IG 14.3.1-9.

14.4 Exceptions in Input-Output

Semantic Ramifications

S1. Note that DEVICE_ERROR is not restricted to READ or WRITE. An implementation may raise this exception for any operation on a device which, for example, has been physically removed.

S2. Note that the RM does not preclude the possibility of an operation encountering situations that permit one of several exceptions to be raised, depending on the implementation — e.g., whether USE_ERROR or DEVICE_ERROR is raised when attempting to write a tape whose write ring has been left off is implementation-dependent. It is implicit in the nature of exceptions that only one will be raised. The RM states, "If more than one error condition exists, then the corresponding exception that appears earliest in the following list is the one that is raised" (RM 14.4/1). This implies that each language-defined exception has a priority assigned to it. Note that implementation-defined exceptions have no priority defined by the language. The priority of each exception should, however, be detailed in Appendix F.

S3. The RM does not restrict the exceptions that each operation may raise to those explicitly listed in the IO_EXCEPTIONS package. In particular, since it seems likely that OPEN and CREATE may dynamically allocate data structures that describe the external file status as well as associated buffers, we have assumed that such procedures may propagate STORAGE_ERROR (RM 11.1/8). Other exceptions may be propagated as well, since no rule in RM 14 forbids this. Full details should be listed in Appendix F.

Changes from July 1982

S4. There are no significant changes.

Changes from July 1980

S5. The 1980 version of the RM did not have an equivalent section.

Exception Conditions

The exceptions discussed here have been covered in the preceding sections (IG 14.2.1/E through IG 14.3.9/E).

Test Objectives and Design Guidelines

The tests for exceptions are covered in the sections describing the operations that raise them.

14.5 Specification of the Package IO_Exceptions

The implications have been discussed in IG 14.4.

14.6 Low Level Input-Output

Semantic Ramifications

S1. The procedures SEND_CONTROL and RECEIVE_CONTROL are totally implementation-dependent (including the types of the parameters).

Changes from July 1982

S2. There are no significant changes.

Changes from July 1980

S3. There are no significant changes.

Test Objectives and Design Guidelines

T1. Check that the procedures provided by the implementation work as specified.

Check that the procedures may be called with parameters of the types specified by the implementation.

ACVC IMPLEMENTERS' GUIDE IMPROVEMENT PROPOSAL

(See Instructions - Reverse Side)

NAME OF SUBMITTING ORGANIZATION

☐

VENDOR

☐

USER

ADDRESS (Street, City, State, ZIP Code)

☐

MANUFACTURER

☐

OTHER (Specify): _____

PROBLEM AREAS

a. Paragraph Number and Wording:

b. Recommended Wording:

c. Reason/Rationale for Recommendation:

NAME OF SUBMITTER (Last, First, MI)

WORK TELEPHONE NUMBER

MAILING ADDRESS (Street, City, State, ZIP Code)

DATE OF SUBMISSION

INSTRUCTIONS:

In a continuing effort to improve the ACVC Implementers' Guide, the ACVC Maintenance Organization provides this form for use in submitting comments and suggestions for improvements. All users of the Implementers' Guide are invited to provide suggestions. In the PROBLEM AREAS block, be as specific as possible about particular problem areas such as wording that is ambiguous or incorrect, and give proposed wording changes that would alleviate the problems. An acknowledgment will be mailed to you to let you know that your comments have been received and are being considered.

Please fold this form along the lines indicated, tape along the loose edges, affix proper postage, and mail it.

(Fold along this line)

(Fold along this line)

AFFIX
POSTAGE
HERE

ASD/SCOL

WRIGHT-PATTERSON AFB OH 45433-6503

ATTN: AMO MANAGER

END

DATE

FILMED

APRIL

1988

DTIC